A GENERAL OBJECT MODEL TRANSFORMATION SYSTEM

THESIS

John P. Mullaney
Captain, USAF

AFIT/GCS/ENG/94D-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-18

A GENERAL OBJECT MODEL TRANSFORMATION SYSTEM

THESIS

John P. Mullaney
Captain, USAF

AFIT/GCS/ENG/94D-18

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE A General Object Model Transformation System	5. FUNDING NUMBERS
6. AUTHOR(S) John P. Mullaney	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/94D-18
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter WL/AAWA Wright Laboratory Wright-Patterson AFB, OH 45433	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Cecil and Fullenkamp developed a program that transforms knowledge captured in an object-model in one environment into a different object-model in a different environment. This program worked well for the intended purpose, but if one of the object-models is replaced by a different object-model then this transformation program has to be re-engineered. As the object-modeling paradigm becomes more prevalent, many systems are experiencing this problem. The primary goal of this research was to determine what reusable knowledge could be extracted from these types of program and used to build a general object-model transformer that generalizes such transformation programs. Toward this end, we analyzed several cases of object-model transformations, designed a generic version of an object-model transformer, and implemented and tested a prototype transformer. The research results provide an approach to analyzing the problem, designing a solution, and transformation templates for various aspects of a solution strategy. These results should be extremely beneficial to software engineers designing object-model transformation systems.

14. SUBJECT TERMS Object-Oriented Databases, Object-Models, ODMG-93	15. NUMBER OF PAGES 86
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

AFIT/GCS/ENG/94D-18

A General Object Model Transformation System

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

John P. Mullaney, B.S.E.E.

Captain, USAF

December 13, 1994

Acknowledgements

This thesis effort has been a unique learning-experience for me. As with most things in life, I didn't do it myself, so there are several people I want to acknowledge for their contributions. First of all, I want to thank the members of my thesis committee, Captain Cecil, Dr. Hartrum, and Major Bailor, for their time and suggestions. Their help made the final product considerably better than it would have been otherwise. In particular, Major Bailor's guidance kept me moving in the right direction. One day, I hope to be able to see the "big picture" as well as he does. I, also, want to thank Dave Doaks and Dan Zambon for the outstanding computer system support they provided us throughout this entire year. Theirs is, by far, the most well-run computer network I have ever been associated with. In addition, I want to thank my database partner Al Harris and the other members of the KBSE research group for all of their support.

I would like to thank my sons, Patrick and Matthew, for reminding me what is really important and for always giving me a good excuse to take a break. Finally and most importantly, I want to thank my wife, Kathy, whose undying faith in me and constant love helped me more than I can say.

John P. Mullaney

Table of Contents

Page

Acknowledgements ii

List of Figures vi

Abstract vii

I. Introduction 1-1

1.1 Background 1-1

1.2 Problem 1-3

1.3 Hypotheses 1-4

1.4 Goals 1-4

1.5 Assumptions 1-5

1.6 Scope 1-5

1.7 Conclusion 1-5

II. Analysis of Object-Model Transformations 2-1

2.1 Introduction 2-1

2.2 Object Models 2-1

2.2.1 OMG Object Model 2-1

2.2.2 ODMG Object Model 2-2

2.3 Object-Model Transformations 2-3

2.3.1 Equivalent Case 2-5

2.3.2 Non-equivalent Case 2-8

2.4 Conclusion 2-18

Page

III. Object-Model Transformer Design 3-1

3.1 Introduction 3-1

3.2 Transformation Description Language 3-3

3.3 Transformation Kernel 3-4

3.4 Object-Model Descriptions 3-7

3.5 Control Algorithm 3-8

3.6 Conclusion 3-9

IV. Implementation of a General Object-Model Transformer 4-1

4.1 Introduction 4-1

4.2 Implementation Plan 4-1

4.3 Implementation of the Object-Model Descriptions 4-3

4.4 Implementation of the Transformation Kernel 4-6

4.4.1 Transform Object 4-6

4.4.2 Transform Relationship 4-9

4.4.3 Transform Aggregate 4-12

4.4.4 Create New Object 4-13

4.4.5 Set New Attribute 4-15

4.4.6 Create New Relationship 4-15

4.4.7 Specific Transformations 4-15

4.5 Implementation of the Controller 4-15

4.6 Conclusion 4-17

V. Validation of the General Object-Model Transformer 5-1

5.1 Introduction 5-1

5.2 Validation Domain 5-1

5.3 Validation Testing 5-3

5.3.1 Equivalent Case 5-3

Page

5.3.2 Flattened Model 5-4

5.3.3 Loss of Multiple Inheritance 5-5

5.3.4 Different Model 5-6

5.3.5 Gaining Hierarchy 5-6

5.3.6 Adding Structure 5-6

5.4 Replacing the Existing Transformation Program 5-6

5.5 Conclusion 5-7

VI. Conclusions and Recommendations 6-1

6.1 Results and Conclusions 6-1

6.1.1 Object-Model Transformation Analysis 6-1

6.1.2 General Object-Model Transformer Design 6-2

6.2 Recommendations for Further Research 6-4

Appendix A. Script for the Prototype General Object-Model Transformer A-1

A.1 Introduction A-1

A.2 Script A-1

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure Page

2.1. A Generic Object-Model 2-5

2.2. A Flattened Version of the Generic Object-Model 2-9

2.3. A Generic Object-Model without Multiple Inheritance 2-11

2.4. A Generic Object-Model 2-13

2.5. A Generic Object-Model 2-15

3.1. A conceptual view of a General Object-Model Transformer 3-2

3.2. General Transformations from Chapter II 3-4

4.1. An Implementation of a General Object-Model Transformer in Software Refinery 4-2

4.2. Meta-Model of Object Models 4-3

4.3. 4-7

4.4. Example for Transform Object 4-8

4.5. 4-10

4.6. 4-14

5.1. A Subset of an OCU Application 5-2

5.2. Instance Diagrams of the Source and Target Model Descriptions 5-3

5.3. Instance Diagrams of the Source and Target Models of a Simple-Light 5-4

5.4. Flatter Object-Model for the OCU Model 5-5

A.1. Instance of a Course to be transformed A-1

AFIT/GCS/ENG/94D-18

Abstract

Cecil and Fullenkamp developed a program that transforms knowledge captured in an object-model in one environment into a different object-model in a different environment. This program worked well for the intended purpose, but if one of the object-models is replaced by a different object-model then this transformation program has to be re-engineered. As the object-modeling paradigm becomes more prevalent, many systems are experiencing this problem. The primary goal of this research was to determine what reusable knowledge could be extracted from these types of program and used to build a general object-model transformer that generalizes such transformation programs. Toward this end, we analyzed several cases of object-model transformations, designed a generic version of an object-model transformer, and implemented and tested a prototype transformer. The research results provide an approach to analyzing the problem, designing a solution, and transformation templates for various aspects of a solution strategy. These results should be extremely beneficial to software engineers designing object-model transformation systems.

A General Object Model Transformation System

I. Introduction

1.1 Background

Object Database Management Systems (ODBMS) are widely recognized as the means to support a range of applications that traditional Relational Database Management Systems (RDBMS) can not (13). In particular, RDBMS do not support the long transactions and complex objects with multiple versions required by applications like Computer Aided Design (CAD) and Computer Aided Software Engineering (CASE) tools. Typically, a user of one of these applications will work for hours on one object, making changes to an engineering design or modifying a program. During the entire time, no one else should be allowed to modify the object, so all of this work must be considered as one transaction. The prototype domain-oriented application composition system being developed by the Knowledge-Based Software Engineering (KBSE) Group requires this same type of database support; therefore, it seemed logical to incorporate an ODBMS into this composition system.

The domain-oriented application composition system provides an environment in which software applications for a particular domain can be composed, or synthesized, using lower-level components of that domain. The heart of this application composition environment is an application composer, called Architect. Architect is an evolving prototype first developed by Anderson and Randour (1, 10) . It runs in the SOFTWARE REFINERY development environment and was written in REFINE which is a wide-spectrum specification language. Architect allows application developers to create a specification for a required application program. These specifications are created by connecting the formal specifications for pre-defined software components, in a correctness preserving manner. Since these components represent natural artifacts within this domain, the developer knows what the components are and understands how they relate to each other. The Architect Visual System Interface (AVSI) (7, 15) presents these components to the application developer as icons which the developer simply connects in an appropriate way to create a new specification. This specification can then be compiled and executed. If it has captured the desired behavior then a formal specification for the application program can be generated; otherwise, the developer can change it and try again.

All of the domain artifacts for the domain currently being manipulated by Architect are represented as objects in the SOFTWARE REFINERY environment. This set of objects can be thought of as the working technology base for Architect. However, this working technology base ceases to exist when the Architect session ends, so any domain artifacts are lost unless they are saved in some type of persistent technology base. The original version of Architect used a system of flat files for this persistent storage. While this system worked, it was inefficient for large domains with many applications. Cecil and Fullenkamp (6) improved on this design by using an ITASCA ODBMS as the persistent technology base.

To incorporate the ODBMS into the domain-oriented application composition environment Cecil and Fullenkamp (6) developed an object-model of the Object Connection Update (OCU) software architecture model that was different than the one used in Architect. The object-model used for Architect was flat because the domain artifacts were stored as files. With an ODBMS, the natural hierarchy of the OCU model can be represented; therefore, the ODBMS implementation can make better use of the OCU model. However, in order to move domain artifacts from Architect into the ODBMS or from the ODBMS into Architect, the domain artifacts must be transformed from one object-model to the other. Cecil and Fullenkamp (6) wrote a set of programs to do these transformations, but their programs encapsulated detailed knowledge about the two models being transformed. Consequently, a new set of programs would need to be written if a different type of object-model was required. The focus of this research was to increase the functionality of the persistent technology base by making these transformation programs more general.

1.2 Problem

The goal of this research was to develop a general object-model transformation system that transfers the knowledge captured in an ODBMS object-model to an abstract syntax tree for REFINE which can be used in the working technology base for Architect. Then as new application domains are required, they could be specified with an object-model and transformed into an object-model for Architect. In the best case, a previously defined specification could be re-used merely by transforming the information from one object-model to another.

In order to transfer knowledge from the ODBMS to Architect's working technology base, Cecil and Fullenkamp (6) had to transform the knowledge captured in an object-model corresponding to their model of the OCU architecture into an abstract syntax tree for Refine which uses an object-model corresponding to a different model of the OCU architecture. This transformation is accomplished by traversing the source object-model from top to bottom creating an object of the correct class in the target model to correspond to each object as it is encountered. Then the new objects in the target model must be recomposed into the proper order. The transformation programs they designed incorporated knowledge of the source and target object-models. For example, these programs include the names of the object classes and are designed to follow the actual structure of these specific object-models. Although these programs work very well, every time the structure of either object-model changes another transformation program has to be written. A more general transformation program would be written at a more abstract level. It would contain as few of the specific details of the object-models as possible. Since these details are required for the transformation, they would have to be given to the transformer as inputs. Then the transformer could instantiate a specific transform program for the required transformation.

1.3 Hypotheses

The following two hypotheses were the primary focus of this research:

1. That there is a set of general transformations that can be used to transform object-models.
2. That it is possible to build a system that can transform object-models based on this set of general transformations.

1.4 Goals

I established the following objectives for this research:

1. Analyze object-model transformations and develop a set of general transformations.

2. Implement these transformations in REFINE.

3. Develop a specification for a general object-model transformation system.

4. Implement and test a general object-model transformation system.

1.5 Assumptions

In order to conduct this research, I assumed I would have the necessary computer resources to develop and test the proposed system.

1.6 Scope

The scope of this effort was limited to developing a transformer that worked between the ITASCA ODBMS and the SOFTWARE REFINERY environment.

1.7 Conclusion

The following chapters describe my approach to solving the problem of transferring knowledge from the ITASCA ODBMS to the working technology base for Architect. In particular, this transfer was accomplished with a general object-model transformation system. This system will transform knowledge from one object-model format to another when given a description of the source and target object-models. The next chapter is an analysis of object-model transformations that results in specifications for six general object-model transformations. These transformations form the kernel for a general object-model transformer. Chapter III describes the high-level design of such a transformer. Chapter IV describes the implementation of a general object-model transformer prototype using SOFTWARE REFINERY, and Chapter V describes the validation testing of this prototype. Finally, Chapter VI discusses the results and conclusions of this effort and suggests some topics for additional research.

II. Analysis of Object-Model Transformations

2.1 Introduction

The heart of an Object-Model Transformer is the set of general transformations that are used to transform one object-model into another. This chapter develops a set of six general transformations by analyzing several cases of object-model transformations. The chapter begins by defining what an object-model is, in particular, by describing the pertinent aspects of two object-model standards. The rest of the chapter provides a case analysis of object-model transformations. Besides the general transformations, this analysis also yields the information needed by a transformer to conduct an object-model transformation.

2.2 Object Models

Object-models are used in many disciplines of computer science and software engineering with many and varied definitions of what an object-model is. The following sections discuss the Object Management Group (OMG) Object-Model and the Object Database Management Group (ODMG) Object-Model. These two object models are related and are widely, although not universally (8), accepted standards for implementing object-models. Consequently, they provide a good working definition of an object-model.

2.2.1 OMG Object Model. The OMG defines a set of features called the Core Object-Model, which is intended to be the core component for object-based system (e.g. Object Request Broker (ORB) or ODBMS) standards. These other object-based system standards are supposed to build on the Core Object-Model by adding any other components that they need to form a standard profile of that type of object-based system. All systems that comply with the standard profile will be interoperable, and all systems of other types that comply with the Core Object-Model will be able to share objects. (14)

The OMG Core Object-Model defines the following characteristics for an object:

1. Objects represent real-world entities, like a student, a book, or a cruise missile
2. Objects have unique identities that do not change during the life of the object
3. Objects have operations which represent their behavior and provide the interface to their state
4. Objects are instances of a specific type, where the type defines the set of common operations that apply to all instances of that type
5. Types can inherit operations from their supertypes

2.2.2 ODMG Object Model. The ODMG has specified a standard for Object Databases which is referred as ODMG-93 (2). ODMG-93 extends the OMG Object-Model by specifying additional components that together with the core component form a profile for ODBMS. In particular, ODMG-93 defines persistent objects, introduces exception handling to operations, introduces queries and transactions, and defines attributes and relationships for objects.

For this discussion, the most important extension is the addition of attributes and relationships since we are primarily concerned with the structure of object-models. The OMG Object-Model does not specify how an object keeps track of its state; it only requires the object to be able to report its state information via its operations. ODMG-93 extends the OMG model by specifying that objects maintain their state in their attributes and relationships, collectively called properties. Attributes are defined for an object type and take literal values. Each attribute has two built-in operations set-value to write its value and get-value to read its value. Relationships are defined between two objects, and can have multiplicity of one-to-one, one-to-many, or many-to-many. Depending on the multiplicity, there are several built-in operations specified for relationships, including create, delete, and traverse.

In addition to establishing a standard object-model, ODMG-93 defines an Object Query Language (OQL) and an Object Definition Language (ODL). OQL is a declarative query language that facilitates access to data stored in ODBMSs that comply with the ODMG-93 Object-Model. As such, OQL is to ODBMSs as SQL is to relational DBMSs. On the other hand, ODL is a specification language for defining interfaces to object types that comply with the ODMG-93 Object-Model. It is programming language independent, and therefore, it is an aid to portability because once a schema is specified in ODL, it can be defined in a variety of programming languages and implemented on any ODMG-93 compliant ODBMS.

For the rest of this discussion, object-models are assumed to comply with the ODMG-93 standard. This will become more important later when the object-model descriptions for the general object-model transformer are specified in ODL.

2.3 Object-Model Transformations

An object-model transformation is a process that changes an object-model described in one syntax into a semantically similar object-model described in a different syntax. For example, the schema for an ODBMS and the internal representation of data within some software application could both be represented by object-models. However, these two object-models may be described in different languages. For example, the ODBMS schema may be written in ODL and the software application may be written in C++. In order to load data from the ODBMS into the software application, the data stored in the ODBMS has to be transformed into an instance of the object-model in the software application.

There are two general cases of object-model transformations, the equivalent case and the non-equivalent case. In the equivalent, case the source and target object-models have the same objects with the same attributes and the same relationships. The non-equivalent case is separated further into two cases.

1. The case in which the resultant model does not contain as much information as the source model therefore the transformation results in a loss of information. Usually, information is lost because the resultant model does not represent the inheritance hierarchy of the object-model as well as the source model does. For example, if the resultant model does not use multiple inheritance or represents the object-model with fewer levels of inheritance (i.e., the model possesses less of a hierarchical structure).
2. The case in which the target model contains more information than the original model therefore the transformation adds information to the object-model. For example, if the target model requires additional structure.

The rest of this chapter analyzes these general cases in order to establish a set of general transformations and to determine what information a general object-model transformer needs in order to automatically transform one object-model into another. Where possible the general transformations are expressed formally, as REFINE transforms. A REFINE

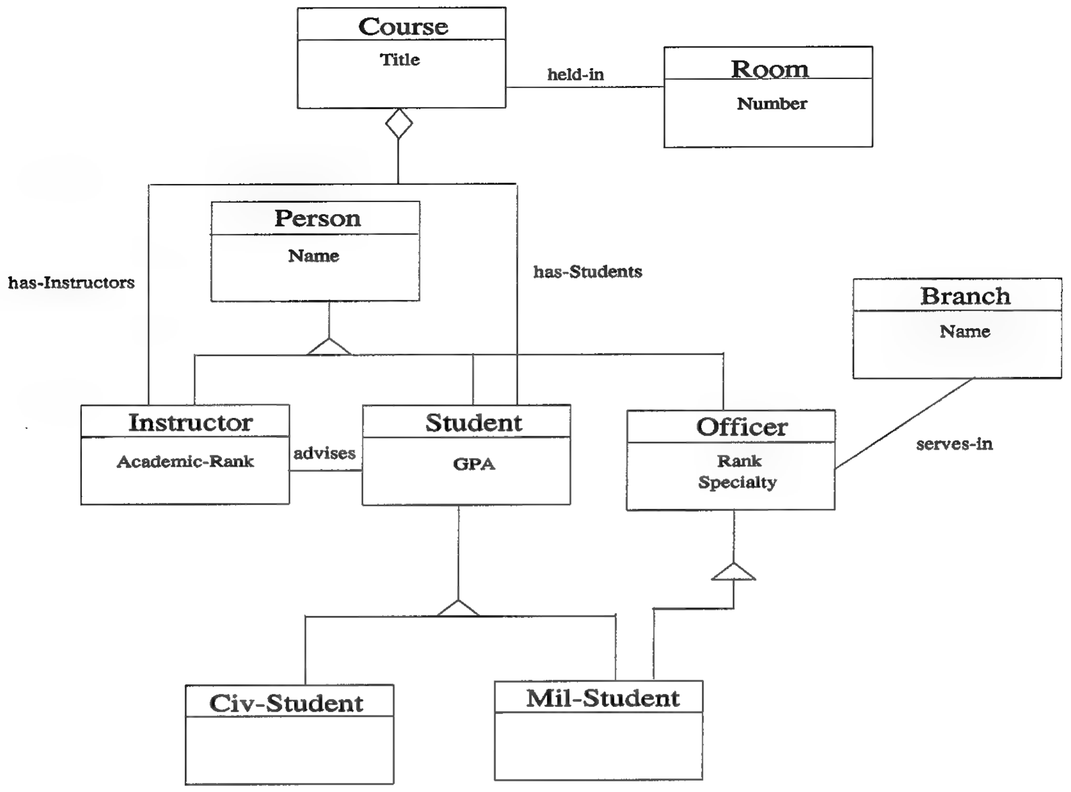

Figure 2.1 A Generic Object-Model

transform specifies a transformation with two predicates separated by a $\longrightarrow$. The predicate on the left side is called a precondition and the predicate on the right is called the postcondition. A transform can be interpreted semantically as, the precondition must be true before the transform can be applied and the postcondition will be true as a result of this transform. (3).

2.3.1 Equivalent Case. In the simplest case, an object-model is transformed into an equivalent object-model which means that for each object in the source model, there is an equivalent object in the target model with the same attributes and relationships. To produce this transformation an object is created in the target model for each object in the source model and its attribute values are set equal to the attribute values of the corresponding object in the source model. Or more formally, this transformation can be specified as a REFINE transform:

```
(T1)  x in source-model & a in attribute-of(x) -->
        class(y) = class(x) & y in target-model & fa(b)(b = a & b(y) = a(x))
```

where x and y are objects and a and b are attributes of x and y respectively. For example, referring to the object-model in Figure 2.1, for each instance of Student in the source model, an instance of Student is created in the target model, and the attribute GPA of the newly created instance is set equal to the value of GPA in the original instance. This procedure is repeated until all objects in the source model are transformed.

Objects which are an aggregation of other objects are a special case because the objects that form the aggregation are transformed first and included in the aggregate. This transformation can be specified in REFINE as:

```
(T2)   x in source-model & a in aggregate-of(x) -->
         y = x & y in target-model & a = b & b in aggregate-of(y)
```

where a, b, x, and y are objects. For example, again referring to the object-model in Figure 2.1, when transforming instances of Course, all instances of Student and Instructor corresponding to a particular instance of Course are created and attached to that instance of Course as part of the transformation. This procedure is repeated until all aggregated objects in the source model are transformed.

Once all the objects are created, any relationships between them must be represented. This is done by making a relationship in the target model for each relationship in the source. For example, the relationship Advises between Instructor and Student is represented in the target model. In REFINE, this transformation can be specified as:

```
(T3)  <x,a,b> in source-relation -->
         y = x and c = a and d = b and <y,c,d> in target-relation
```

where x is the name of the relation, a is the name of one of the classes involved in the relation, and b is the name of the other class involved in the relation. Note this equation assumes that all relations will be binary, though it is conceivable that some relations may be n-ary. In order to make this equation more general, a sequence of object names could replace a and b in the ordered tuple.

Thus there are three general transformations that are required to transform on object-model into an equivalent object-model. Analysis of the example above reveals that the transformer executing these transformations needs the following information.

- What classes compose the source object-model and which of these classes are aggregations
- What classes compose the target object-model
- What are the attributes for each class in the source model
- What are the attributes for each class in the target model
- What is the mapping from the attributes in the source to the attributes in the target if there is more than one attribute of the same type, e.g. symbol, string, etc.

- What relationships exist between the classes in the source model
- What relationships exist between the classes in the target model
- What is the mapping between the relationships in the source and the relationships in the target
- What is the multiplicity of these relationships

2.3.2 Non-equivalent Case. The last section discussed the case in which the target model is equivalent to the original model. However, in most cases the source object-model is different from the target model. The following sub-sections discuss different cases of non-equivalent transformations and explain when the general transformations described for the equivalent case are sufficient. When these transformations are not sufficient, new transformations are developed if appropriate. First, the case in which the transformation results in a loss of information is discussed; then, the case in which the transformation results in adding information to the object-model is discussed.

2.3.2.1 Losing Information. A transformation results in a loss of information if the target model does not have as much structure or semantic meaning as the original model. The following paragraphs discuss three cases of transformation in which information is lost. The first case occurs when the target model does not have as many levels of inheritance as the source model, i.e., the target model is "flatter" than the source model. In the second case the source model uses multiple inheritance and the target model does not. In the third case, the objects in the target model do not have all the attributes or relationships that the objects in the source model do.

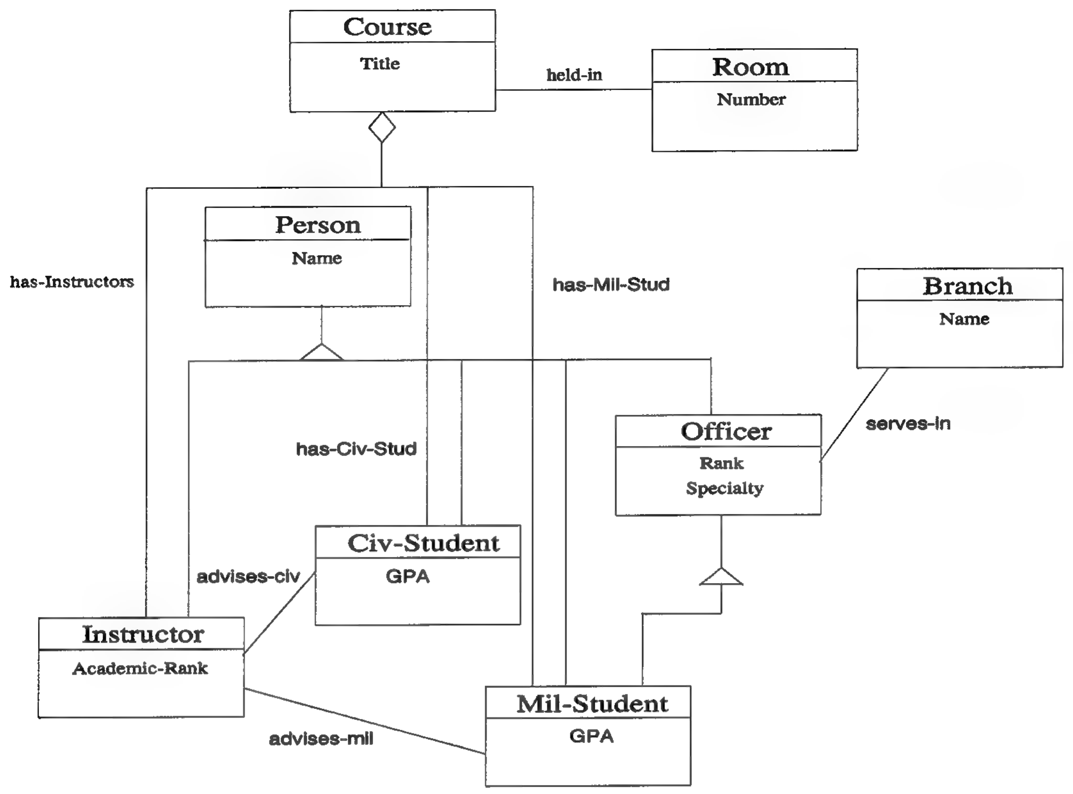

Figure 2.2 A Flattened Version of the Generic Object-Model

Case 1: *Transforming into a flatter model.* An object-model is said to be a flatter model than another object-model when it does not have as deep of an inheritance hierarchy. For instance, the object-model in Figure 2.2 is similar to the object-model in Figure 2.1, but it is a flatter model because Civ-Student and Mil-Student do not inherit from Student. Transforming an instance of the object-model in Figure 2.1 into this flatter model results in a loss of information since the fact that Civ-Student and Mil-Student are specializations of Student is not represented. However, objects of type Civ-Student and Mil-Student still have the same attributes as they do in Figure 2.1 because the attributes that are inherited from Student in Figure 2.1, for example GPA, are explicitly attributes of Civ-Student and Mil-Student in Figure 2.2. Since the objects in the flatter model look the same as the objects in the source model, the procedure for transforming the objects is exactly the same as the one for transforming into an equivalent model. Therefore, specification T1 describes these object transformations. Transforming the relationships is more difficult, though. While all the objects in the flatter model have the same number of relationships, these relationships are not strictly the same as the relationships in the source model. For example, the Advises relationship is inherited by Civ-Student and Mil-Student from Student in Figure 2.1, but in Figure 2.2, the relationship between Instructor and Civ-Student is called Advises-Civ since Civ-Student does not inherit from Student. In order to perform this transformation, the transformer has to know that Advises maps to Advises-Civ; therefore, specification T3 needs to be modified to include this mapping. In REFINE, this modified transformation can be specified as:

```
(T3a)    <x,a,b> in source-relation -->
            y in target-map(x) & c = a & d = b & <y,c,d> in target-relation
```

where x is the name of the relation, a is the name of one of the classes involved in the relation, and b is the name of the other class involved in the relation, and target-map is the set of relationships in the target that correspond to the given relationship in the source.

Case 2: *Transforming into a model that does not have multiple inheritance.* Transforming an object-model that uses multiple inheritance into one that does not is similar to transforming into a flatter model. In both cases, some objects lose a level of inheritance. In fact, the general transformations that describe transforming into the flatter model describe transforming into a model without multiple inheritance. For

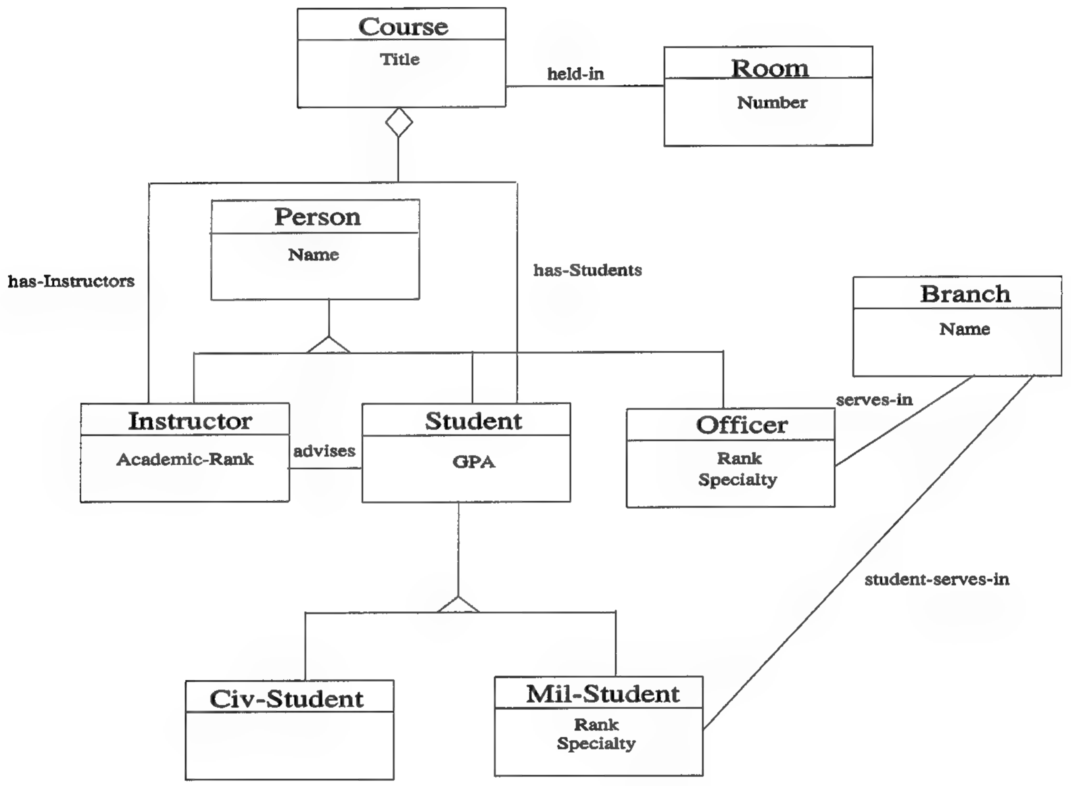

Figure 2.3 A Generic Object-Model without Multiple Inheritance

example, Figure 2.3 is equivalent to Figure 2.1 except that it does not support multiple inheritance. In particular, Mil-Student inherits from both Student and Officer in Figure 2.1; while in Figure 2.3, Mil-Student only inherits from Student. Once again, an object of type Mil-Student has the same attributes in both the source and the target models; therefore transforming from the source to the target is the same as in the equivalent case. However, as with the case of the flattened model, any inherited relationships in the source model that are not inherited relationships in the target model, for example Student-Serves-In, must be mapped to the relationship it corresponds to.

Case 3: *Transforming into a different model.* The past two sections have discussed transforming an object-model into another object-model in which the inheritance hierarchy was different but the objects being transformed were identical. Many times it is desirable to transform an object-model into another in which the objects are different. Such a transformation is more difficult to generalize because the transformer has to figure out which attributes and relationships are the same between the two objects and which are missing. As an example, the object-model represented in Figure 2.4 could be a possible target for the model in Figure 2.1. Notice that the attributes of Officer are different in the two models. In particular, Rank is in both objects, but Specialty and Serves-In are in the source object and not in the target object. Thus to transform an instance of Officer in the source to an instance of Officer in the target, the transformer would have to set Rank equal to Rank and disregard Specialty and Serves-In. This transformation is still described by specification T1 since the right hand side includes the predicate

```
fa(b)(b=a and b(y) = a(x))
```

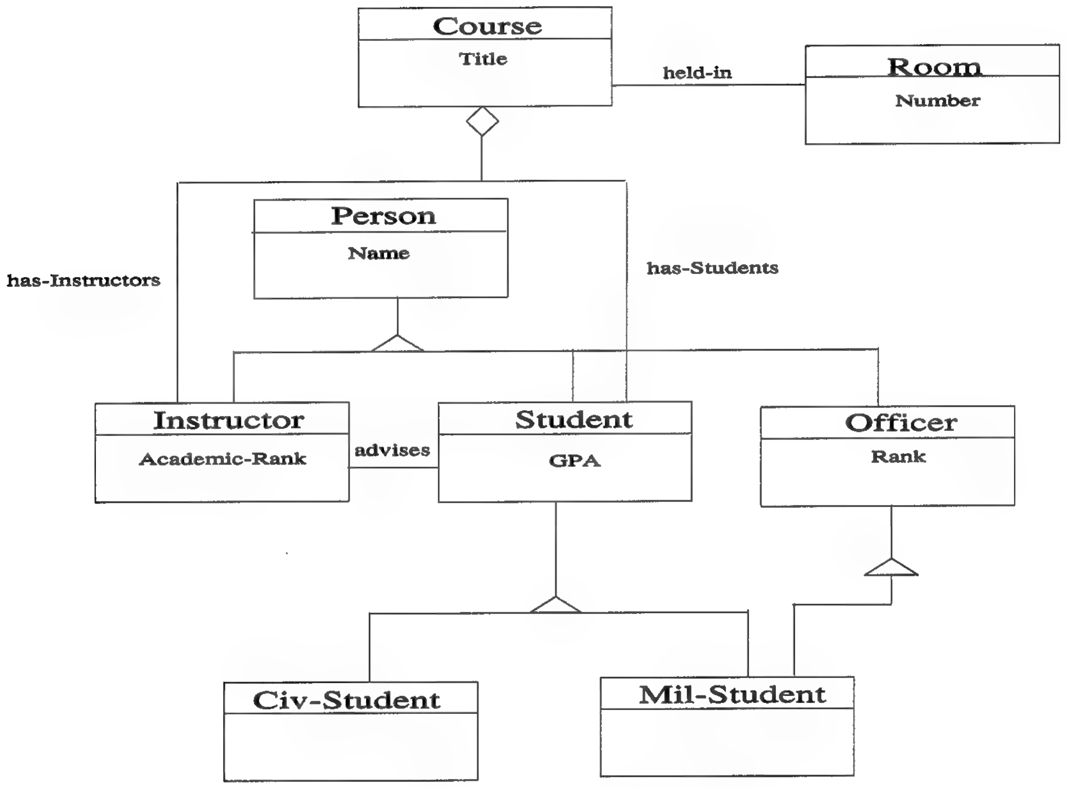

Figure 2.4 A Generic Object-Model

where b is an attribute of the target object and a is an attribute of the source object. This predicate can be interpreted as meaning that if an attribute is not in the target model then disregard it. This is a subtle point of which an implementer of this transformation must be aware.

Required Information for Losing Transformations. Since the losing transformations require the same transformations as the equivalent case with a modified transformation for relationships, the information required to do these transformations is the same as in the equivalent case with the addition of a mapping from relationships in the source to the relationships in the target.

2.3.2.2 Gaining Information.

A transformation results in a gain of information if the target model has more structure or semantic meaning than the original model. This type of transformation could occur if the target model has more layers of inheritance or multiple inheritance, if some of the objects in the target have attributes or relationships that the source model does not have, or if the target has some object classes that the source model does not. Gaining transformations are more troublesome than losing transformations because the added information has to come from somewhere. The following paragraphs discuss these cases.

Case 1: *Transforming into a Deeper Model.* Transforming an object-model into another object-model that has more levels of inheritance, e.g., from Figure 2.2 to Figure 2.1 or from Figure 2.3 to Figure 2.1, is essentially the same as transforming from the model with more inheritance to the model with less inheritance. That is, first

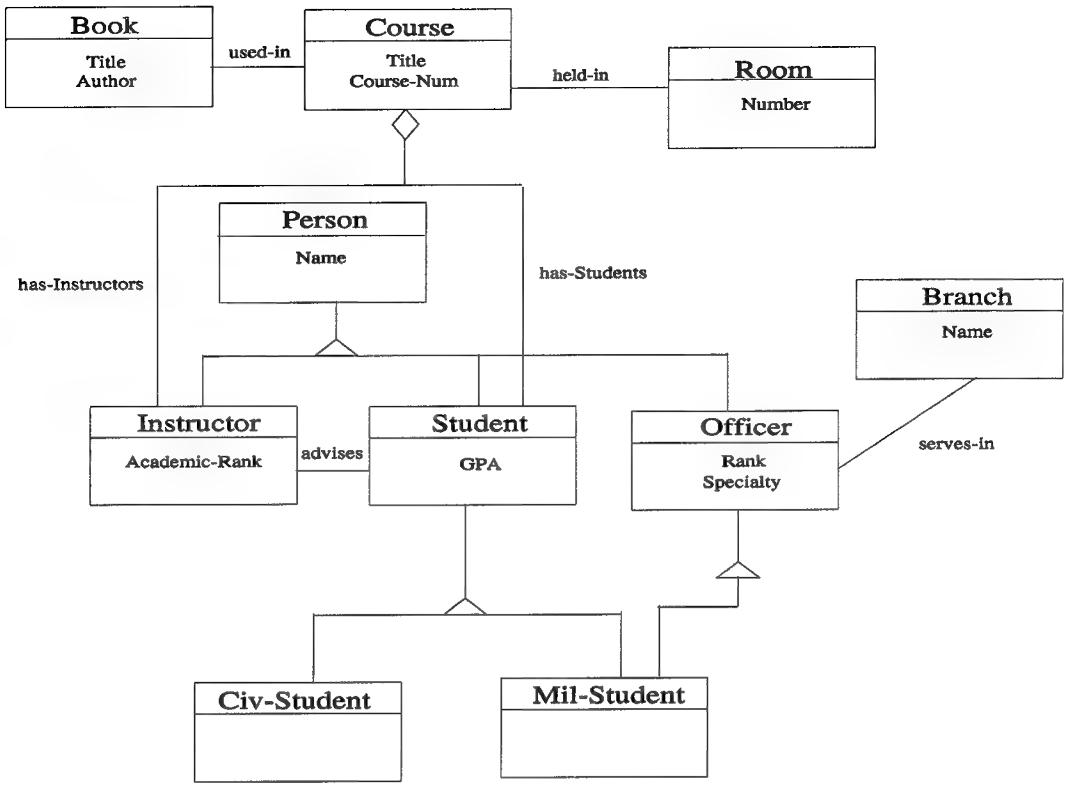

Figure 2.5 A Generic Object-Model

transform each object then transform the relationships between them using a mapping to determine which relationships from the source correspond to which relationships in the target. However, this transformation is making a more profound change to the object-model than the other way does. In case 1 before, where we went from more hierarchy to less hierarchy, some information is lost about an object, but the meaning of the transformed object is preserved. However, in the case of less to more hierarchy, meaning is being added to the transformed object. While in some cases this transform is useful, there are other cases where the new meaning is incorrect and may cause a failure in the target application.

Case 2: *Adding an Object.* Another way that information can be added to an object-model is by adding an object. For example, the object-model in

Figure 2.5 is similar to the object-model in Figure 2.1 except that the model in Figure 2.5 has a new type called Book. To transform the model in Figure 2.1 to this model the transformation is the same as in the other cases with the addition of a transform to create a new object in the target that does not exist in the source. More formally, this transformation can be specified as a REFINE transform:

```
(T4)  x in target-model & x ~ in source-model -->
        make-object(x) & (y in attributes(x) --> set-attrs(x,y,value(y)))
```

where x is the class of the new object and y is an attribute of x. To accomplish this transformation the transformer has to know what x is, what x's attributes are, and where to find the values of these attributes.

Case 3: *Adding an Attribute.* In some cases the difference between the source model and the target model is that some of the objects in the target model have more attributes than their counterparts in the source model. For example, Course in Figure 2.5 has Course-Num that Course in Figure 2.1 does not have. In order to do this transformation, a transformation that sets the value of an attribute is required. This can be specified as a REFINE transform:

```
(T5)  x in source-model & y in target-model & class(x) = class(y) &
      b in attribute-of(y) & b ~ in attribute-of(x) -->
        set-attrs(y, b, b-value)
```

where x and y are objects, b is the attribute that is being set, and b-value is a legal value for b. To accomplish this transformation the transformer has to know what x and y are, what b is, and where to get b-value.

Case 4: *Adding a Relationship.* In some cases, the target model has additional relationships between objects that are not in the source model. For example, Used-In in Figure 2.5 is not in Figure 2.1. To accomplish this transformation, the transformer needs a transform that creates a relationship. More formally,

```
(T6)   yXz --> <X,y,z> in relations of Target
```

where X is a relationship and y and z are the objects involved in the relationship. To do this transformation, the transformer has to know what X, y, and z are, and how to determine which y relates with which z. In most cases this will be difficult to determine automatically.

Required Information for Gaining Transformations. In order to perform gaining transformations, a transformer needs all the information required for the equivalent and losing transformations plus the values of any attributes which are not provided by the source model and the descriptions of any relationships in the target model that are not in the source model. On a more fundamental level, a transformer needs to know that the information being added as a result of the transformation is, at least, not incorrect. Ideally, the added information makes for a truer representation of reality.

2.4 Conclusion

As a result of the preceding case analysis of object-model transformations, it is apparent that there are six general transformations that characterize object-model transformations. The first three transformations described how objects and relationships are transformed. These transformations apply when the object-models being transformed are equivalent or when the target model has weaker semantics than the source model. They will also apply to some cases where the target model has stronger semantics. The last three transformations describe how objects, attributes, and relationships are added to an object-model. These transformations apply in the cases where the target model has some additional structure than the source model.

These six transformations have been formally described using REFINE syntax, and some or all of these transformations must be incorporated in any object-model transformation program. An additional product from this analysis is a list of information required to do object-model transformations. This information is required in the next chapter to design a general object-model transformer.

III. Object-Model Transformer Design

3.1 Introduction

This chapter describes a top-level design of a general object-model transformation system which is strongly influenced by the knowledge gained from the analysis of object-model transformations in Chapter II. This design is implementation-independent, and as such, provides a starting point for developing general object-model transformers for many different environments. Chapter IV provides a description of one such implementation in SOFTWARE REFINERY with an ODBMS as the source environment. This chapter begins by describing a general object-model transformer on a conceptual level. This high-level description is followed by a discussion of the major aspects to the design of a general object-model transformer.

A general object-model transformer is a software system that transforms many different types of object-models to create another object-model that is more convenient for a particular application. These transformations are accomplished by applying a set of general transformations to the objects and relationships in the object-model. Figure 3.1 is a conceptual view of a general object-model transformer. The heart of this system is a transformation kernel containing a set of general transformations. Conceptually, an object-model transformer works as follows. The system inputs a description of the source and target object-models. Next, it uses these descriptions to determine which transformations from the transformation kernel are needed to transform each object. If a required transformation is not in the transformation kernel, it is provided by the user using the transformation description language. Once the required transformations are selected, the

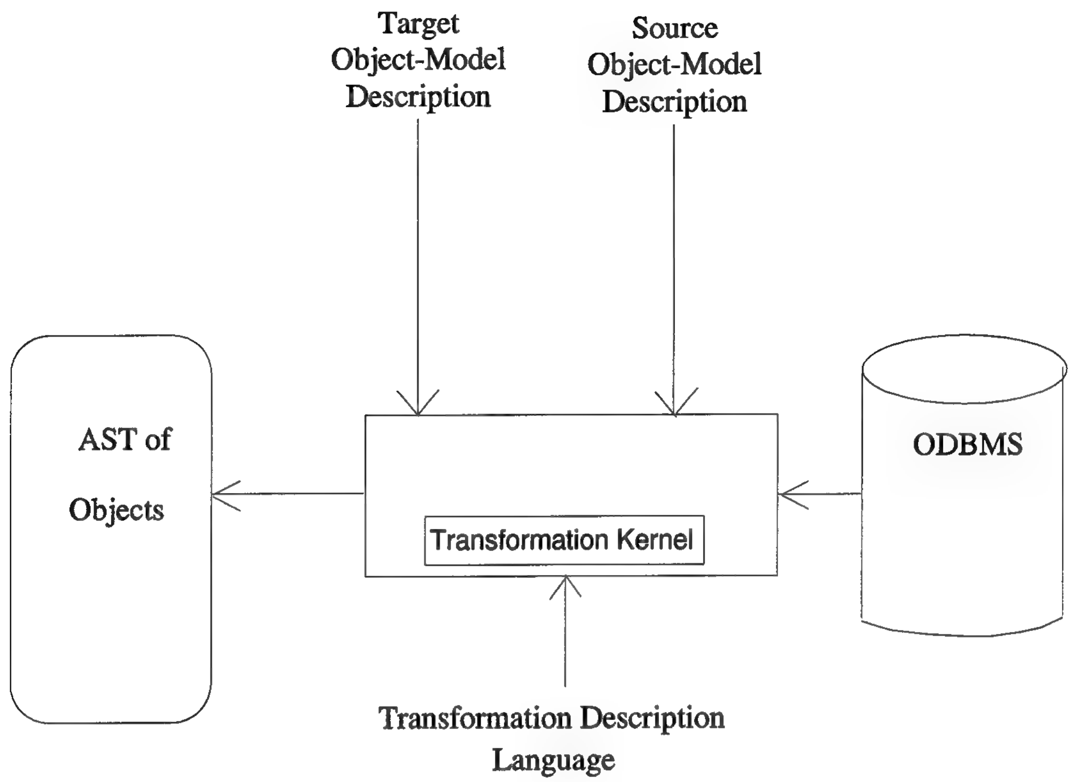
Target Object-Model Description
Source Object-Model Description
AST of Objects
Transformation Kernel
ODBMS
Transformation Description Language

Figure 3.1 A conceptual view of a General Object-Model Transformer

system composes a control algorithm that applies the transformations in the appropriate order. Finally, the system executes the control algorithm to traverse an instance of the source object-model and transform it into an instance of the target object-model.

There are four aspects to the development of the general object-model transformer.

1. Developing the transformation description language.
2. Determining how to implement the general transformations.
3. Determining how to describe the source and target object-models to the transformer.
4. Determining what control algorithm to use.

The following sections discuss these four aspects in more detail.

3.2 Transformation Description Language

A transformation description language is a language for specifying object-model transformations. Basically, this language contains a set of commands for manipulating objects and object-model descriptions. These commands can be instantiated with the appropriate binding for the environment in which the transformer is executing. With a transformation description language, the developer implementing an object-model transformer does not have to worry about the details of the interface between the transformer and the source or target model environments. In addition, a set of transformations written in a transformation description language are more portable to other environments because once a new set of bindings are written, the transformations should work as before.

As a minimum, a transformation description language should have commands to do the following.

- Create an object of an arbitrary class.
- Access an object given its object identifier.
- Read an attribute of an object.
- Set an attribute of an object.
- Create a relationship between two objects.
- Get a set of all objects involved in a relationship.
- Access all objects of a particular class.
- Get the description of a particular class from the object-model description.
- Get a list of the attributes of a particular class from the object-model description.

```
                    T1

x in source-model & a in attribute-of(x) -->
   class(y) = class(x) & y in target-model & fa(b)(b = a & b(y) = a(x))

                    T2

x in source-model & a in aggregate-of(x) -->
   y = x & y in target-model & a = b & b in aggregate-of(y)

                    T3a

<x,a,b> in source-relation -->
   y in target-map(x) & c = a & d = b & <y,c,d> in target-relation

                    T4

x in target-model & x ~ in source-model -->
   make-object(x) & (y in attributes(x) --> set-attrs(x,y,value(y)))

                    T5

X in source-model & y in target-model & class(x) = class(y) &
b in attribute-of(y) & b ~ in attribute-of(x) -->
   set-attrs(y, b, b-value)

                    T6

yXz --> <X,y,z> in relations of Target
```

Figure 3.2 General Transformations from Chapter II

- Get the description of a relationship from the object-model description.

3.3 Transformation Kernel

The specifications for general transformations developed in Chapter II (Figure 3.2) provide the basis for the transformation kernel. These specifications are implemented in an executable transformation description language. The following are some considerations for implementing these general transformations.

- These transformations can be kept as general as possible by parameterizing all the variant parts. For example, the identity of the object being transformed is going to change each time the transformation is executed. The variant parts of each general transformation in Figure 3.2 are described below.

- T1 - Transform an object
  * Source object class
  * Target object class
  * Object identifier for the source object
  * Source attribute names
  * Target attribute names
  * Mapping of source names to target names
- T2 - Transform an object composed of aggregates
  * All items from T1
  * Name of the attribute that contains the aggregate objects
- T3a - Transform a relationship
  * Name of the source relationship
  * Names of the target relationships
  * Object identifier of the object that implements the relationship
  * Classes of the objects involved in the relationship
  * Names of the target objects involved in the relationship
  * Mapping of source names to target names
- T4 - Create an object
  * Class of the object

        * Attributes of the object

    - T5 - Set an attribute

        * Name of the attribute

        * Object identifier of the object to which the attribute belongs

        * Value of the attribute

    - T6 - Create a relationship

        * Name of the relationship

        * Object identifiers of the objects involved in the relationship

- In some cases transformations may be composed from other transformations. For example, since T2 requires transforming objects, T1 could be used by T2.

While the implemented transformations in the transformation kernel are as general as possible, they still encapsulate some design decisions based on assumptions that are not true for every conceivable object-model transformation. In cases where these assumptions do not hold, the implemented transformations cannot accurately transform the object-model, and the user has to specify a unique transformation. Note that this unique transformation will still conform to one of the general transformation specifications in Figure 3.2 – only the way it is implemented is unique. Any unique transformations should be specified in the transformation description language and added to the transformation kernel.

3.4 Object-Model Descriptions

For the transformer to be able to transform an object-model, it must have a useful description of the structure of the source and target object-models. This description is used by the general transformations and the control algorithm. Such a description clearly shows what objects there are, what attributes these objects have, and what relationships exist between these objects. The most convenient way to do this is to parse in a file-based description of each object-model, and transform it into some type of intermediate representation useful to the transformer.

The Object Definition Language (ODL), described in Chapter II, is a logical choice for the file-based description of the object-models. ODL is becoming the standard for describing ODBMS schemas and most applications of an object-model transformer are going to involve some sort of ODBMS as either the source or the target environment. In addition, ODMG-93 specifies a Backus-Naur Form description of ODL which could readily be used with a parser generator tool to develop a parser for ODL. The big advantage to using a standard language like ODL for the object-model description language is portability. Once an object-model is described in ODL, it can be used by several different object-model transformers.

The intermediate representation of the source and target object-models depends on the environment in which the transformer is executing. In most cases, an abstract syntax tree structure based on a meta-model of object-models would be the most useful representation. This type of structure is easy for a parser to build and easy for the transformer to manipulate.

3.5 Control Algorithm

The most difficult part of the object-model transformer is developing the control algorithm. The control algorithm is the procedure that the transformer follows in order to transform an object-model. This procedure specifies which object to start with, which transformation to use on each object, and what strategy to follow for picking the next object to transform, e.g., bottom-up, top-down, random-selection, etc. A control algorithm developer must look at the source and target object-models and decide which transformations to apply to each object and in what order. These decisions are difficult because they are based primarily on the semantics of the object-model. They are doubly difficult when the semantics are captured in the meaning of the English words used to describe the objects and the relationships between them. For example, a *producer* and a *source* could easily be the same thing in two different object-models. This meaning is particularly difficult for a machine to understand.

In order to develop the control algorithm, the developer has to look at the source and target object-models to determine which strategy for transforming objects to use. This determination is based on the structure of the object-models, e.g. does it have a natural tree structure. Either the source or the target model can drive this decision, depending on which one has less flexibility. (A lack of flexibility might be caused by the environment in which the object-model is implemented.) Once a basic strategy has been picked, the developer has to decide which object should be transformed first if it is not already determined by the transformation strategy. Next, the developer has to look at each object and decide which of the transformations, including specific transformations, will be able to transform it into the target model.

The ultimate object-model transformer should incorporate an automatic control algorithm developer. This control algorithm developer would be implemented with an inference engine based on a knowledge base of decision rules for developing control algorithms for object-model transformations. However, in a less than ultimate design, the designer of the transformer has to build a control algorithm into the transformer. Then the transformer executes this algorithm using the object-model descriptions and the transformation kernel. Whenever it needs to know a piece of information, it queries the user by presenting a pop-up window with a mouse-sensitive menu of the possible responses.

3.6 Conclusion

A general object-model transformer depends first and foremost on the set of general transformations specified in Figure 3.2. The implementation of these transformations becomes the transformation kernel which is used to transform the objects in the source model into the target model. To accomplish this transformation, the transformer first parses ODL descriptions of the source and target models and forms an internal representation of both object-models. Next, it adds any specialized transformations to the transformation kernel. Then following the order specified in the control algorithm, it transforms an instance of the source object-model into an instance of the target object-model. Chapter IV describes a sample implementation using SOFTWARE REFINERY of a general object-model transformer for SOFTWARE REFINERY and the ITASCA ODBMS.

IV. Implementation of a General Object-Model Transformer

4.1 Introduction

This chapter describes the implementation of a prototype of a general object-model transformation system based on the top-level design discussed in Chapter III. This prototype was implemented in SOFTWARE REFINERY for two reasons. First, SOFTWARE REFINERY provides an ideal environment for implementing a general object-model transformer prototype. Second, one of the goals of this research was to generalize the object-model transformation system between the ITASCA ODBMS and the Architect system which is implemented in SOFTWARE REFINERY. The chapter begins by describing the Implementation Plan for a general object-model transformer in SOFTWARE REFINERY; then the intermediate representation of the object-model descriptions, the implementation of the transformation kernel, and the implementation of the controller are discussed.

4.2 Implementation Plan

Figure 4.1 is a conceptual view of a prototype transformer that transforms an object-model stored in the ITASCA ODBMS into an abstract syntax tree for Architect. In this prototype, the controller, which implements the control algorithm, is written in REFINE, and the transformation kernel is implemented as a set of REFINE functions. The controller executes these functions to transform the object-model. Any unique transformations are provided to the transformer as REFINE functions. The object-model descriptions of the source and target models are provided to the transformer in Object Definition Language

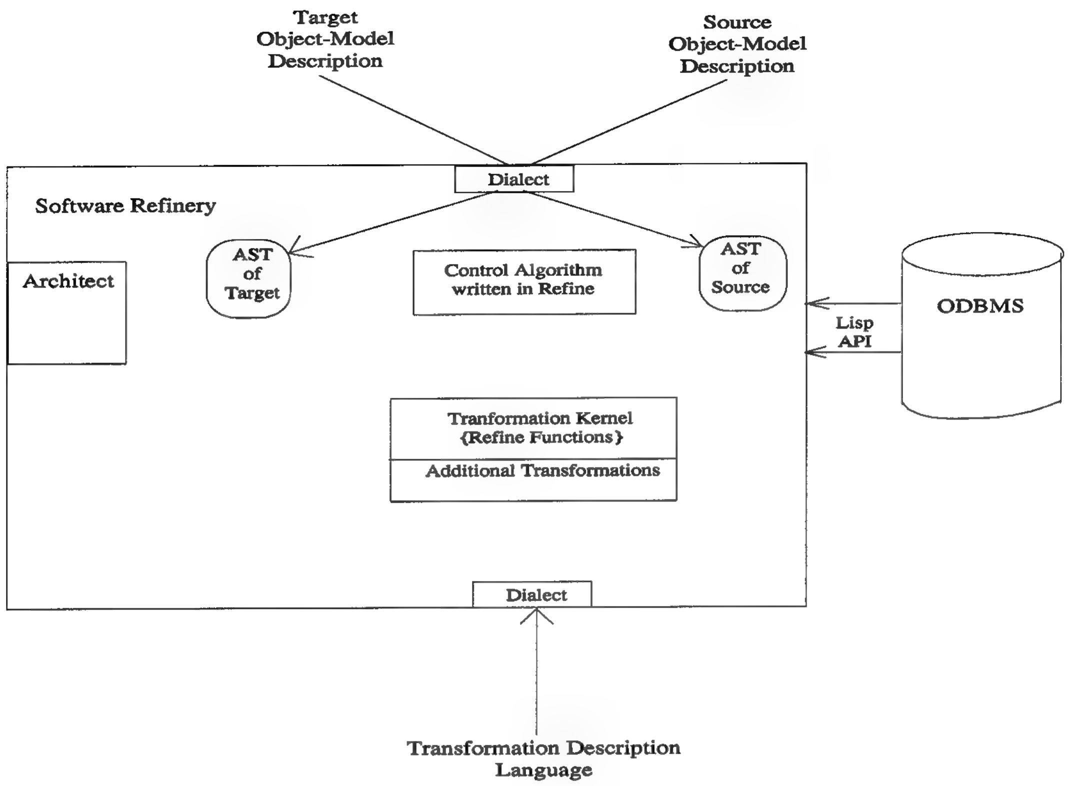
Target Object-Model Description
Source Object-Model Description
Dialect
Software Refinery
Architect
AST of Target
Control Algorithm written in Refine
AST of Source
ODBMS
Lisp API
Tranformation Kernel {Refine Functions}
Additional Transformations
Dialect
Transformation Description Language

Figure 4.1 An Implementation of a General Object-Model Transformer in Software Refinery

(ODL). These descriptions are parsed into the environment and represented as abstract syntax trees which conform to the transformer's meta-model for object-models.

This prototype operates as follows. First the descriptions of the source and target object-model structures are built and any unique transformations are added to the transformation kernel. (The object-model descriptions currently have to be built manually since the ODL parser has not been implemented.) Then the controller follows the control algorithm to accomplish the transformation using the object-model descriptions and the transformation kernel. User input is required whenever the controller cannot determine something automatically.

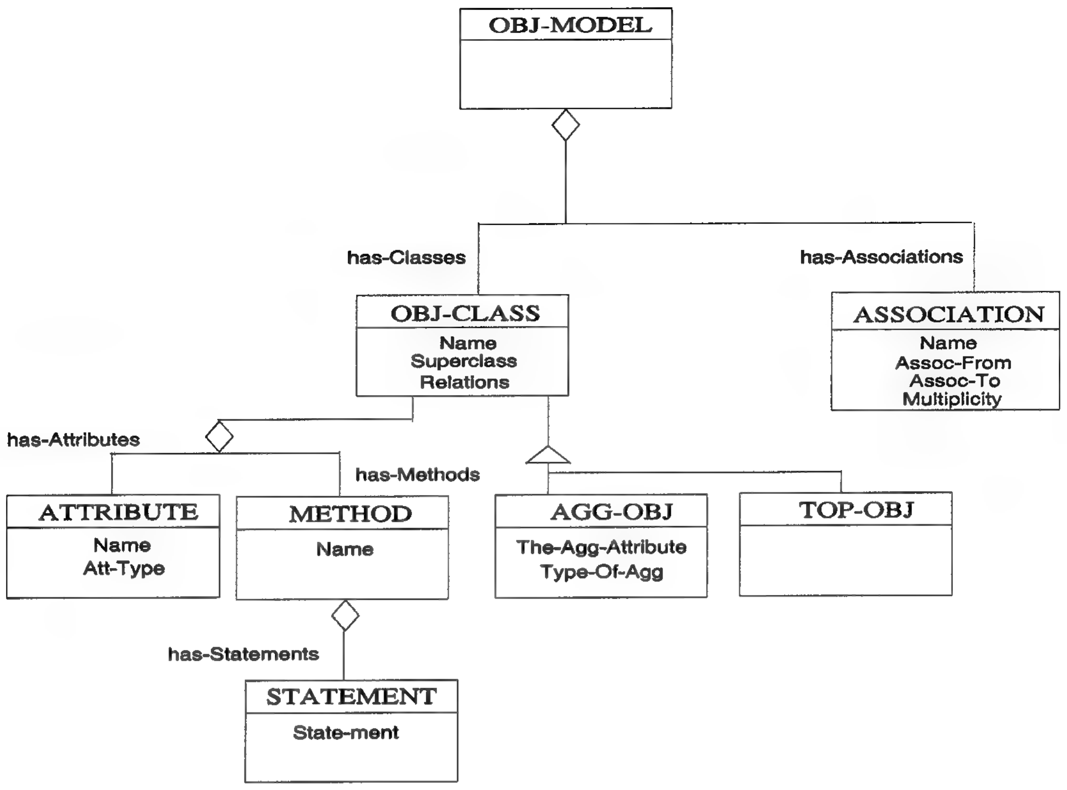

Figure 4.2 Meta-Model of Object Models

4.3 Implementation of the Object-Model Descriptions

The descriptions of the source and the target object-models are used by the controller and the transformation kernel to guide the transformation process. These descriptions are represented in the transformer as abstract syntax trees (AST) in the REFINE object base. ASTs are easy to build with the DIALECT parser and easy to manipulate with REFINE because it has many built-in language features that support tree traversal. Since ASTs in REFINE are specified as object-models, the object-model descriptions are actually represented as object-models. An object-model of an object-model, such as this, is called a meta-model. Thus the source and target object-model descriptions are captured as ASTs which are instances of the meta-model of object-models. Figure 4.2 provides a Rumbaugh

(12) style description of this meta-model. Since the ASTs were intended to be constructed by the ODL parser which was not fully implemented, they were created manually for testing of the transformer.

According to the meta-model, an OBJECT-MODEL is composed of OBJECTS and ASSOCIATIONS with OBJECTS being composed of ATTRIBUTES and METHODS. The model also shows two sub-types of OBJECTS, TOP-OBJECTS and AGGREGATE-OBJECTS, which are useful for the general object-model transformer. The following are descriptions of the object-classes that make up the meta-model.

- OBJ-MODEL - The OBJ-MODEL represents an object-model. An OBJ-MODEL is composed of OBJ-CLASS and ASSOCIATION objects.
- ASSOCIATION - An ASSOCIATION represents a relationship between two OBJ-CLASS objects. ASSOCIATION has four attributes:
    1. Name - the name of the relationship
    2. Assoc-From - the name of the OBJ-CLASS that implements the relationship as one of its attributes. In an environment that implements relationships differently, either of the OBJ-CLASS objects could go here.
    3. Assoc-To - the name of the OBJ-CLASS that is captured in the attribute of Assoc-From.
    4. Multiplicity - a boolean that is true if the multiplicity of the relationship is other than one-to-one.

- OBJ-CLASS - An OBJ-CLASS represents an object class, or type, in the object-model. An OBJ-CLASS is composed of ATTRIBUTE and METHOD objects and has the following three attributes:

  1. Name - the name of the class
  2. Superclass - a set of the names of the direct superclasses of this class
  3. Relations - a set of the names of the ASSOCIATION objects which are implemented by this OBJ-CLASS

- TOP-OBJ - The TOP-OBJ is the OBJ-CLASS that is the top-level object in the OBJ-MODEL. Not all object-models have a top-level object.

- AGG-OBJ - An AGG-OBJ represents an OBJ-CLASS that is composed of aggregate objects. An AGG-OBJ has the following two additional attributes:

  1. The-Agg-Attribute - contains the name of the ATTRIBUTE that is a set of aggregate objects. For example, if a Book is composed of Chapters, the Book is an AGG-OBJ and The-Agg-Attribute would equal Has-Chapters.
  2. Type-Of-Agg - contains the type (i.e. AGG-OBJ or OBJ-CLASS) of the objects in the set of aggregate objects. For example, if a Book is composed of Chapters and a Chapter is composed of Pages, then Book and Chapter are of type AGG-OBJ. Furthermore, The-Agg-Attribute of Book would equal Has-Chapters and the Type-Of-Agg of Book would equal AGG-OBJ, since Chapter is of type AGG-OBJ.

- ATTRIBUTE - An ATTRIBUTE represents an attribute of an object. Class ATTRIBUTE has the following attributes:

   1. Name - the name of the attribute

   2. Att-Type - the type of the attribute, i.e. string, integer, etc.

- METHOD - A METHOD represents a method of an object. A METHOD is composed of STATEMENT objects, and has a Name attribute which represents the name of the method.

- STATEMENT - A STATEMENT represents a single statement in a METHOD. A STATEMENT has an attribute called State-ment which is a string representing the statement.

4.4 Implementation of the Transformation Kernel

In order to implement the transformation kernel for the general object-model transformer prototype, it was necessary to implement each of the general transformations developed in Chapter II. In keeping with the implementation plan, these transformations were designed assuming the ITASCA ODBMS contained the source object-model and the REFINE object-base contained the target object-model. For this prototype, REFINE was used as the transformation description language, so each of the transformations was implemented as a REFINE function. The following paragraphs describe the implementation of the general transformations.

4.4.1 Transform Object. The first transformation, specification T1, transforms an object in the source model into an object in the target model. The function Transform-

```
function Transform-Object (Object-Class: Object,
                           Source-Name : any-type,
                           Name-List   : set(tuple(symbol,symbol))) =

 let (Source-Attribute   = 'nil,
      Target-Attribute   = 'nil,
      Source-Class       = Name(Object-Class))
 let (Target  = make-object(Get-Target-Name(Source-Class, Name-List)))

  (enumerate Next-Attribute over has-Attributes(Object-Class) do
     Source-Attribute <- Name(Next-Attribute);
     Target-Attribute <- Get-Target-Name(Source-Attribute, Name-List);
     set-attrs(Target,
               Target-Attribute, itasca::send(Source-Attribute, Source-Name)));
   Target
```

Figure 4.3

Object, Figure 4.3, implements this general transformation. It accepts as parameters an Object Identifier (OID), a class description , and a name map. The OID is the unique tag to the object in the ODBMS that is being transformed. The class description is the object in the meta-model description for the source that represents the class of the object being transformed. The name map maps the class and attribute names of the source class to the class and attribute names of the target class. The name map makes the transformation more general by eliminating the requirement that the names in the source and target model be identical.

The function transforms the object corresponding to the OID as follows. First, it determines the class of the source object from the class description then it uses the name map to determine which class this corresponds to in the target model. Armed with the class of the target object, the function creates an object of that class in the target model. Then using the object class description, it enumerates over the attributes of the source class. For each attribute, it gets that attribute's value from the ODBMS, determines which target attribute the attribute corresponds to, and sets the target attribute to the correct value.

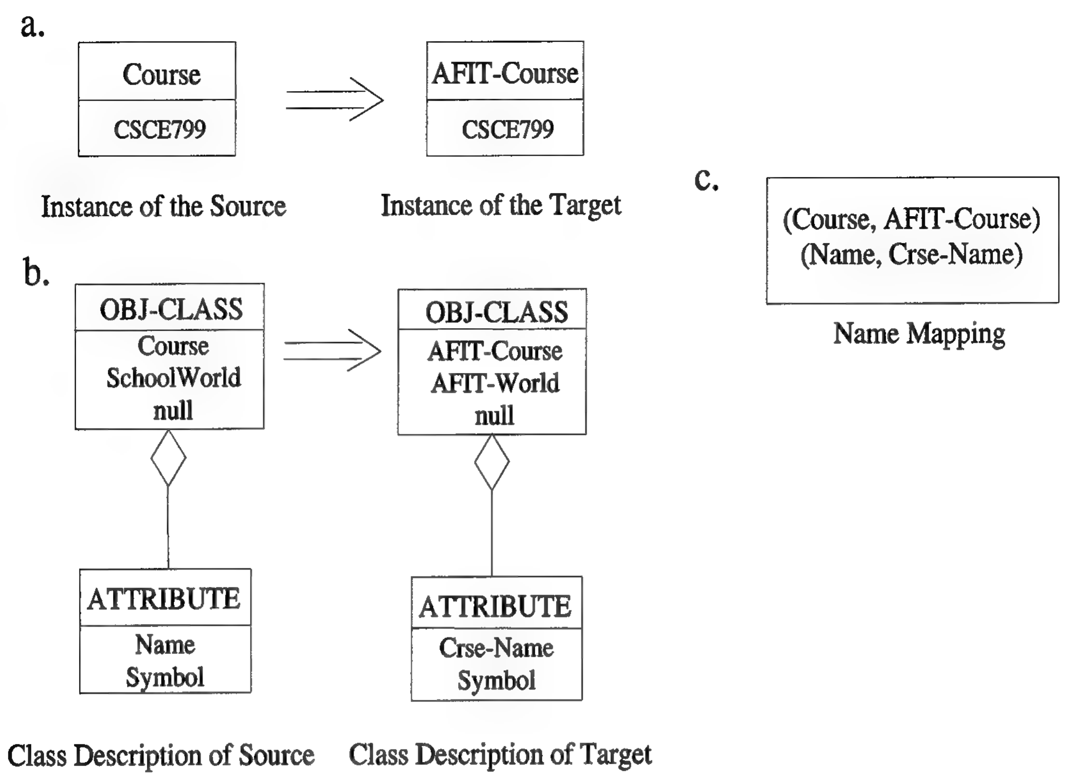

Figure 4.4 Example for Transform Object

This implementation of specification T1 is based on the following assumptions.

1. The Target model has a class that corresponds to the source class, i.e.

```
class(x) in source --> (ex y)(class(y) in target  class(y) = class(x)).
```

2. The Target object has an attribute of the same type that corresponds to each attribute in the Source object, i.e.

```
(fa a)(a in attributes-of(x)  class(x) in source) -->
        (ex b)(b = a  type(b) = type(a)  b in attributes-of(y)
          class(y) = class(x)  class(y) in target).
```

For example, assume we need to transform the instance of Course CSCE799 shown in part a of Figure 4.4, into an instance of AFIT-Course. Transform-Object is passed the Object-Class description object shown in part b of Figure 4.4, the database OID for Course CSCE799, and the Name-List shown in part c of Figure 4.4. The class of Course CSCE799 is Course and from the Name-List it is apparent that Course maps to AFIT-Course. So an instance of AFIT-Course is created in SOFTWARE REFINERY. The only attribute of class Course is Name which maps to the Crse-Name attribute of AFIT-Course. Therefore the Crse-Name attribute of the newly created instance of AFIT-Course is set to CSCE799. Thus Course CSCE799 has been transformed into AFIT-Course CSCE799.

4.4.2 Transform Relationship. The function Transform-Relation, Figure 4.5, implements the general transformation in specification T3a. This function transforms a relationship between two objects in the ITASCA ODBMS into an equivalent relationship in the REFINE environment. This function accepts as parameters:

- a description of the relationship,
- the OID of the source object that implements the relationship,
- a set of objects that are to be included in this relationship,
- a mapping of names in the source to names in the target,
- a mapping of the relationships in the source to relationships in the target.

The description of the relationship is the association object from the object-model description of the source model (see Figure 4.2) that corresponds to the relationship to be transformed. This description provides the name of the relationship, its multiplicity, and the

```
function Transform-Relation (Assoc        : Object,
                             Source-OID   : any-type,
                             Object-Set   : set(object),
                             Name-List    : set(tuple(symbol,symbol)),
                             Relation-Map: set(tuple(symbol, symbol, re::binding, symbol))) =


 let  (From-Class                    = Assoc-From(Assoc),
       The-Relation                  = Name(Assoc),
       Relation-is-a-Set?            = Multiplicity(Assoc),
       New-Relation    : symbol      = nil,
       Rel-Obj         : object      = nil)
let (Target = Find-Object('user-object,itasca::send('name, Source-OID)),
       New-From-Class = Get-Target-Name(From-Class, Name-List))

    (if Relation-is-a-Set? then

      (enumerate x over Object-Set do
        New-Relation <- Get-Relation-Name(The-Relation, New-From-Class,
             Instance-Of(x), Relation-Map);
        set-attrs(Target,
         New-Relation, Retrieve-Attribute(Target, Find-Attribute(New-Relation))
                        with x))

    else         % the Relation is one to one and is not a set
      Rel-Obj  <- arb(Object-Set);
      New-Relation <- Get-Relation-Name(The-Relation, New-From-Class,
           Instance-Of(Rel-Obj), Relation-Map);
      set-attrs( Target,
           New-Relation, Rel-Obj));

    Target
```

Figure 4.5

names of the classes involved. The OID is a handle to the object in the ITASCA ODBMS that contains the relationship as one of its properties. The set of objects contains the objects in the REFINE object base that form the transformed relationship. It is necessary to pass this set as a parameter since the objects are transformed before Transform-Relation is called and it is difficult to access objects in the REFINE object base if they are not named objects. The name mapping is the same as in Transform-Object above; it maps the names of the source classes to the names of the target classes. The relationship mapping accounts for the possibility that relationships in the source may become more than one relationship in the target, for example in the case where the target model is flatter than the source model. The relationship map returns the name of the target relationship given the name of the source relationship and the names of the target classes involved in the relationship.

Transform-Relation works as follows. First, it finds the object in the REFINE object base that corresponds to the object identified by the OID, i.e., the transformed object. If the multiplicity of the relationship is one-to-many, then for every object in the set of objects, Transform-Relation finds the name of the target relationship which the object belongs to and adds it to that relationship for the transformed object. Finding the name of the target relationship for each object is necessary since one relationship in the source model may be represented by multiple relationships in the target and the objects in the set of objects do not necessarily belong to the same target relationship even though they did in the source model. Similarly, if the multiplicity of the relationship is one-to-one, Transform-Relation finds the name of the target relationship which the object belongs to and sets that relationship equal to the object.

This implementation is based on the following assumptions.

1. The Target object-model has a relationship that corresponds to the Source relationship.
2. All objects in the Target relationship already exist.
3. Relationships in the Source and Target models are implemented in the same way, in particular as sets of objects.

4.4.3 Transform Aggregate. The function Transform-Aggregate, Figure 4.6, implements the general transformation in specification T2. This function transforms an object composed of aggregates in the ITASCA ODBMS into an equivalent object in the REFINE environment. Transform-Aggregate accepts the same parameters as Transform-Object. It begins processing by calling Transform-Object with the object in the ITASCA ODBMS that corresponds to Source. This transformation sets all the attributes except the aggregate attribute, i.e., the one that contains a set of other objects. Next, this function gets the set of aggregate objects that belong to the Source object. If these objects are composed of aggregates themselves, Transform-Aggregate is called recursively to transform them. Otherwise, Transform-Object is called to transform them. In either case the aggregate attribute is set equal to the set of transformed aggregate objects.

As a result of the way in which aggregation is implemented in the ITASCA ODBMS and in REFINE, this function is equivalent to using Transform-Object followed by Transform-Relation. Except, it is more limited since it can only handle one relationship per object. Therefore for this transformation (from ITASCA to REFINE) , it generally makes more sense to use Transform-Object and Transform-Relation. In other transformations where aggregates are implemented differently than relationships, a Transform-Aggregate function would be more useful.

This implementation of specification T2 is based on the following assumptions.

1. The Target model has a class that corresponds to the source class, i.e.

```
class(x) in source --> (ex y)(class(y) in target  class(y) = class(x)).
```

2. The Target object has an attribute of the same type that corresponds to each attribute in the Source object, i.e.

```
(fa a)(a in attributes-of(x)  class(x) in source) -->
          (ex b)(b = a  type(b) = type(a)  b in attributes-of(y)
           class(y) = class(x)  class(y) in target).
```

3. Aggregation is implemented the same way in the Source and Target models, in particular as sets of objects.

4. Only one type of aggregate for each object.

4.4.4 Create New Object. The general transformation in specification T4 can be implemented with the standard REFINE language constructs for creating an object and setting its attributes. Namely, the

```
make-object(x) and  set-attrs(a,b,c)
```

constructs, where x is an object class, a is an object of type x, b is the name of an attribute of a, and c is the value of b.

```
function Transform-Aggregate (Object-Class: object,
                              Source      : any-type,
                              Name-List   : set(tuple(symbol,symbol))) =

 let  (The-Class      = Name(Object-Class),
       Agg-List       : seq(any-type) = [],
       Agg-Set        : set(object) = {},
       Target         = Transform-Object(Object-Class, Source, Name-List))

    Agg-List <- itasca::send(The-Agg-Attribute(Object-Class), Source);

   (if Agg-Obj(Find-Object('user-object, Type-Of-Agg(Object-Class))) then
      (enumerate Next-Agg over Agg-List do
        Agg-Set <- Agg-Set with
       Transform-Aggregate(Find-Object('user-object, Type-Of-Agg(Object-Class)),
                        Next-Agg, Name-List))
   else
      (enumerate Next-Agg over Agg-List do
        Agg-Set <- Agg-Set with
       Transform-Object(Find-Object('user-object, Type-Of-Agg(Object-Class)),
                        Next-Agg, Name-List)));

    (set-attrs( Target,
     Get-Target-Name(The-Agg-Attribute(Object-Class),Name-List), Agg-Set));

    Target
```

Figure 4.6

4.4.5 Set New Attribute. The general transformation in specification T5 can be implemented with the standard REFINE language construct for setting attributes,

```
set-attrs(x,y,z),
```

where x is an object, y is an attribute of x, and z is the value of y.

4.4.6 Create New Relationship. Since relationships are best implemented as attributes in REFINE, the general transformation in specification T6 can be implemented with the standard REFINE language construct for setting attributes,

```
set-attrs(x,y,z),
```

where x is an object, y is an attribute of x, and z is a set of objects.

4.4.7 Specific Transformations. Each specific transformation is written in the transform description language which is REFINE. Basically, the user writes a REFINE function that implements the required transformation. In most cases, one of the general transformations can serve as a template for this specific transformation, or the general transformations as they are implemented can be used as building-blocks which can be composed to form the specific transformation. Once the new transformation is written the user has to modify the controller to use it.

4.5 Implementation of the Controller

The controller for this prototype object-model transformer attempts to deduce as much information as it can then it asks the user for help as required. The basic algorithm is to transform the top-level object then to recursively transform any objects it has relationships with. Then once all the objects involved in a particular relationship are transformed, the controller transforms the relationship. This algorithm assumes that all the objects in the source object-model can be accessed via relationships from previously transformed objects. If this assumption is not correct, the user will have to change the controller to implement a better control algorithm for that situation.

First, the controller checks the source object-model description to determine if there is a TOP-OBJ. If there is, the controller begins the transformation with this class. Otherwise, it traverses the source object-model description and prints all the OBJ-CLASS Names and asks the user which class to begin with. Once it knows which class to begin with, the controller queries the ITASCA ODBMS for a list of the names of all the objects of that class, presents the user with this list, and asks which object to transform. Next, the controller traverses the target object-model description and presents the user with a list of the classes in the target model and asks which one corresponds to the source object which was just selected. Based on this response and the source class selected above, the controller adds this source class-target class tuple to the name map.

Now that the controller knows which object to transform and which class to transform it into, the name map for the attributes must be constructed. To do this, the controller traverses the ATTRIBUTE objects that correspond to the source OBJ-CLASS and the target OBJ-CLASS and for each attribute in the source model the controller presents the user with a list of the attributes in the target model, and asks which one corresponds to the source attribute. Based on these responses the controller builds the name map. Then it calls Transform-Object to transform the source object based on the description of the object class, the handle to the object, and the name map.

After transforming the object the controller determines if it is involved in any relationships. If so, it presents the user with a list of the classes in the target, and asks which class corresponds to the source class involved in the relationship with the top-level object. Based on this response, it builds a name map like before and transforms the objects keeping track of each transformed object. Then it presents the user with a list of the relationships in the target model, and asks the user which relationships correspond to the source relationship. Based on this response, it builds a relationship map. Then it calls Transform-Relationship to transform the source relationship based on the relationship description, the handle to the object that implements the relationship, the set of transformed objects, the name map, and the relationship map. The controller continues in this way until the entire source model is transformed.

4.6 Conclusion

This chapter described the implementation of a general object-model transformer prototype for the SOFTWARE REFINERY environment. A meta-model of object-models that specifies the abstract syntax tree used as the intermediate representation of the object-model descriptions was also described. Next, the implementation of the general object-model transformations using REFINE as the transformation description language was explained. Finally, the controller was described. The controller is written in REFINE and executes the control algorithm. The next chapter describes the validation of this prototype and discusses some of its limitations.

V. Validation of the General Object-Model Transformer

5.1 Introduction

The general object-model transformer prototype was developed following an evolutionary development methodology. At each stage of the development a prototype was built and tested. Based on the results of the testing, the prototype was refined. Then new functionality was added, and the new prototype was tested and refined. First, the general transformations were developed in this manner. Then the controller which uses these transformations was evolved. The prototypes were validated with the Logic Circuits domain developed by Anderson and Randour and the School domain used in Chapter II. This chapter describes the validation domain in more detail, then discusses the validation testing in which test cases were executed for each case in the case analysis of object-model transformations discussed in Chapter II.

5.2 Validation Domain

Two validation domains were used for the validation of the general object-model transformer prototype. The primary validation domain was the set of OCU Applications from the Logic Circuits domain. A Rumbaugh (12) diagram of the ITASCA ODBMS schema for OCU Applications is shown in Figure 5.1. The secondary validation domain was the set of Courses in the School domain. The schema for a Course is shown in Figure 2.1. The secondary domain was required because OCU Applications did not contain all the necessary constructs required for some test cases; in particular, an OCU Application does not make use of multiple inheritance. Also, the School domain is smaller and therefore

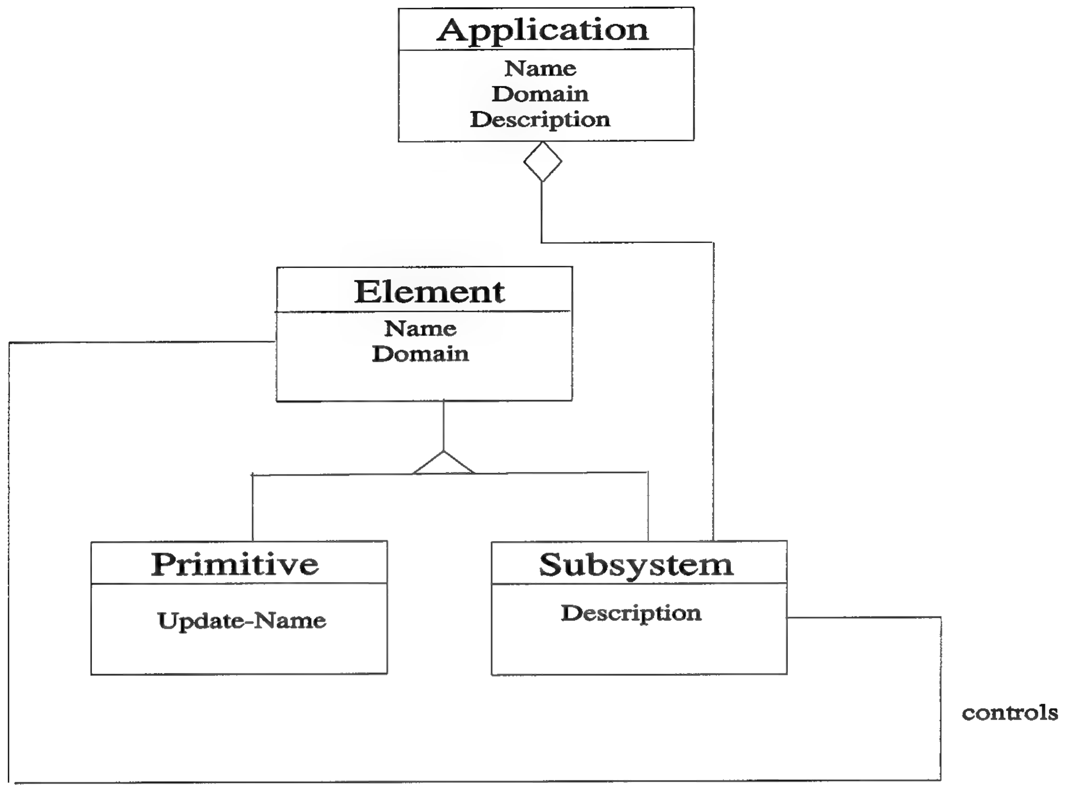

Figure 5.1 A Subset of an OCU Application

more flexible. As a result, it was more convenient to use during the initial phases of evolutionary development. Test cases were validated against both domains where possible.

The OCU model, developed by the Software Engineering Institute (9) was used as the software architecture for Architect. A good description of the OCU Architecture and the way it was implemented in the ODBMS is provided by Cecil and Fullenkamp (6). For this validation testing, a subset of an OCU Application was used. As Figure 5.1 shows an Application is composed of Subsystems. Subsystems control Elements which can be either Primitives of the application domain or other Subsystems.

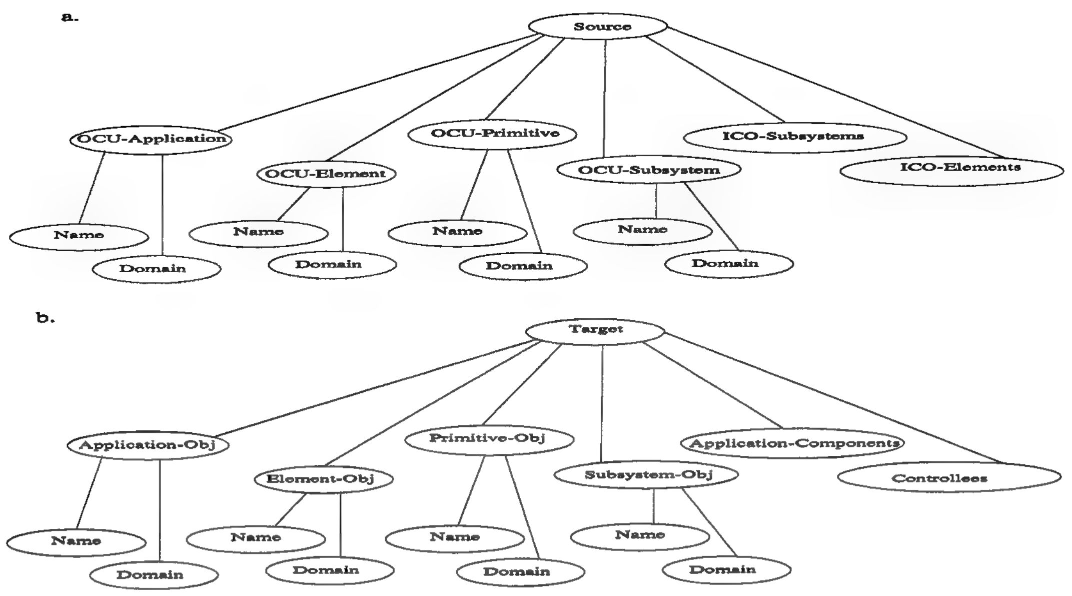

Figure 5.2 Instance Diagrams of the Source and Target Model Descriptions

5.3 Validation Testing

The objective of the validation testing was to ensure object-models in the ITASCA ODBMS could be transformed into object-models in SOFTWARE REFINERY. Separate validation tests were accomplished for each case of object-model transformation. A test case was developed, and the expected results were derived manually before the test was executed. In this way, the actual results of each test could be validated against the expected results.

5.3.1 Equivalent Case. In order to test for the equivalent case, an object-model that was equivalent to the schema of an OCU application was specified in SOFTWARE REFINERY since there was not one currently specified. Next, AST descriptions of the source and newly created target models were built in SOFTWARE REFINERY. These descriptions were constructed by creating an instance of the meta-model of object-models, from Chap-

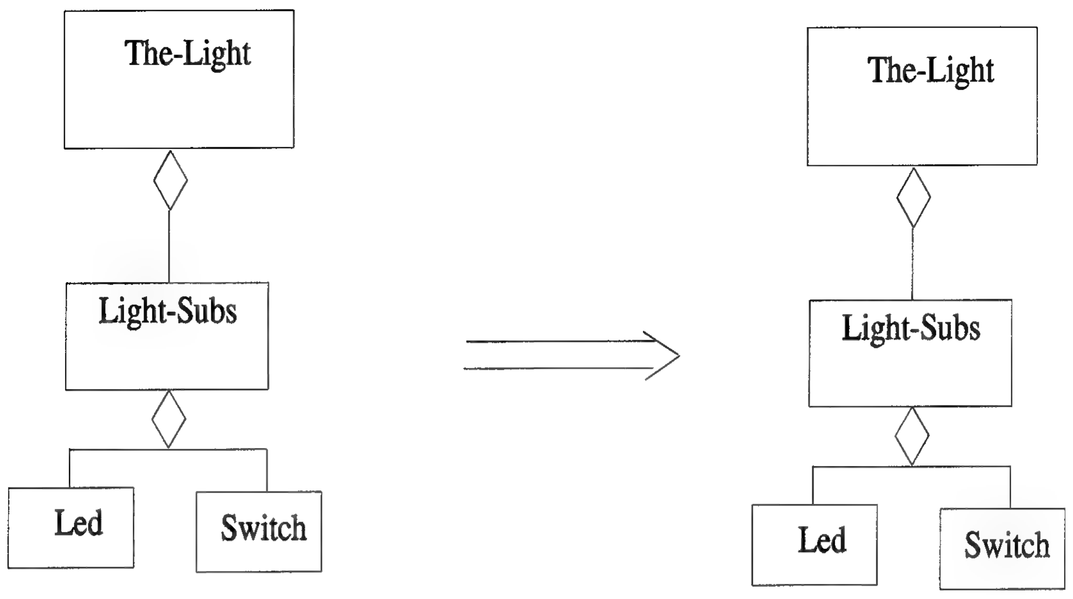

Figure 5.3 Instance Diagrams of the Source and Target Models of a Simple-Light

ter IV, for each object-model (see Figure 5.2 parts a and b). Then a test Application was created in Architect and saved to the ODBMS. Figure 5.3 illustrates this transformation for one sample test Application, namely a Simple-Light. Once everything was in place, the transformer was passed the object-model descriptions and told to transform the Simple-Light. After the transformation the resultant Simple-Light was validated against the expected results.

5.3.2 Flattened Model. The procedure for the flattened model was similar to the equivalent case. First, a flattened version of an OCU Application, Figure 5.4, was specified in SOFTWARE REFINERY, and then a description of this flattened object-model was built. The object-model description for the source model is the same as before. Figure 5.3 illustrates the transformation of the Simple-Light for this case. Notice that the instance diagrams are the same as for the equivalent case. Once again, the transformer was passed

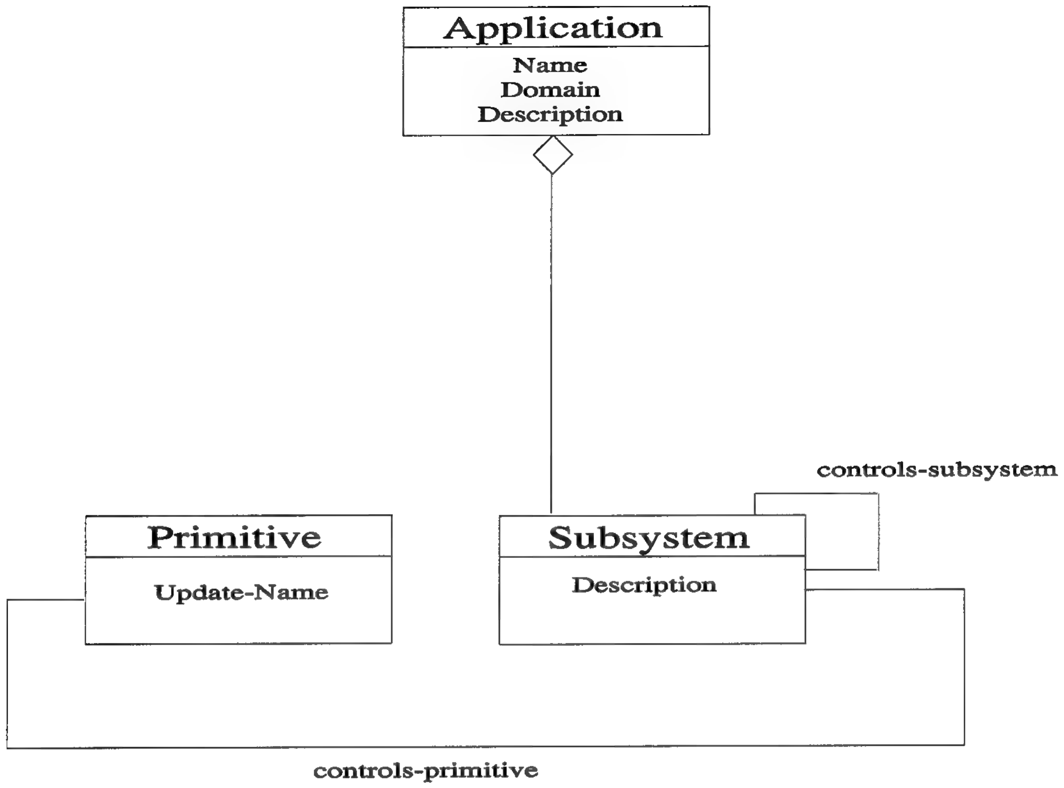

Figure 5.4 Flatter Object-Model for the OCU Model

the object-model descriptions and told to transform the Simple-Light, and the results were validated against the expected results.

5.3.3 Loss of Multiple Inheritance. This case was tested with a Course from the School domain. An OCU Application could not be used since OCU Applications do not use multiple inheritance. In this case, a Course that did not incorporate multiple inheritance was specified in SOFTWARE REFINERY. Then the descriptions for the source and target models were built manually. Since Architect does not compose Courses, a test Course was built in the ODBMS manually. Once everything was ready, the transformer executed with the descriptions as input parameters, and the results were compared with the expected results.

5.3.4 Different Model. This case was tested using the School domain. First, a Course that incorporated a different Officer type was specified in SOFTWARE REFINERY. Then the description for this target model was built manually. Next, using the test Course developed before, the transformer executed the transformation with the source and target model descriptions, and the result was validated against the expected results.

5.3.5 Gaining Hierarchy. In order to test this case, the School domain schema had to be modified in the ODBMS. The Student class was removed so that Civ-Student and Mil-Student inherited directly from Person and new relationships were developed to replace Advises and Has-Students. Then an object-model for the School domain with Student as a superclass was introduced in SOFTWARE REFINERY. Now the School domain in the ODBMS was flatter than the one in SOFTWARE REFINERY. Next the descriptions for the source and target models were built. Since the schema was modified a new test Course had to be built in the ITASCA ODBMS manually. Finally, the test Course was transformed and the resultant model was validated against the expected results.

5.3.6 Adding Structure. As a result of the decision to implement the last three transformations as REFINE language constructs, validating that they work was a matter of testing to see if an object could be created and it attributes set to specified values.

5.4 Replacing the Existing Transformation Program

One of the original goals of this research was to replace the transformation system developed by Cecil and Fullenkamp with a general object-model transformation system. Their transformer transformed objects in the ITASCA ODBMS into objects in SOFTWARE

REFINERY that Architect can manipulate. While the general transformer prototype developed in this research could accomplish this transformation, I chose not to implement it with the limited time available. The difference between the ODBMS schema and the Architect object-model is such that the prototype requires a unique control algorithm and specific transformations for each object. In some cases this may be necessary; however, it would be of little benefit for this research since there is already a unique transformer for this transformation that works well. And a general transformer with a unique set of transformations and control algorithm would still have to be modified whenever the source or target object-model was changed.

5.5 Conclusion

This chapter described the process used to validate the general object-model transformer prototype. As a result of this testing the object-model transformer prototype was validated for all cases using relatively simple models. However, it is not yet complete enough to replace the transformation system developed by Cecil and Fullenkamp. The next chapter discusses the results and conclusions of this research and makes several recommendations for continued research.

VI. Conclusions and Recommendations

Cecil and Fullenkamp (6) developed a program that transforms knowledge captured in an object-model in one environment into a different object-model in a different environment. This program worked well for the intended purpose, but if one of the object-models is replaced by a different object-model then this transformation program has to be re-engineered. As the object-modeling paradigm becomes more prevalent, many systems are experiencing this problem. The primary goal of this research was to determine what reusable knowledge could be extracted from these types of program and used to build a general object-model transformer that generalizes these transformation programs. Toward this end, I analyzed several cases of object-model transformations, designed a generic version of an object-model transformer, and implemented and tested a prototype transformer. The results of this research should be beneficial for a software engineer who is designing an object-model transformer because they provide an analysis of the problem, a design for the solution, as well as templates of the various parts of the solution.

6.1 Results and Conclusions

6.1.1 Object-Model Transformation Analysis. Based on the analysis of object-model transformations, I developed six general transformations that characterize the process of transforming object-models. These general transformations can be stated generally as:

1. make a new object,
2. set an object's attribute,

3. create a relationship between two objects,

4. transform an object into a new object,

5. transform the relationship between two objects into a relationship between two new objects,

6. transform an object with aggregate objects into a new object with aggregate objects.

These general transformations, when implemented in an appropriate high-order programming language and instantiated with the specific details of the object-models being transformed, are the heart of an object-model transformer. Generic versions of these transformations that take specific object-model details as parameters are required for a *general* object-model transformer.

6.1.2 General Object-Model Transformer Design. In order to build a general object-model transformer, a designer has to make four high-level design decisions. These decisions depend on the source and target environments and what the object-models being transformed look like. These decisions are:

1. How to implement the general transformations - The general transformations, above, have to be implemented in the transformation description language. These implementations take the object-model descriptions of the source and target models as parameters.

2. How to describe the object-models to the transformer - To accomplish the transformation the transformer has to have a detailed description of the source and target

object-models. This description has to be represented in a way that is convenient for the transformer.

3. Which control algorithm to use - The control algorithm has to be implemented in the appropriate language and should be designed such that it can successfully transform a large proportion of the object-models that are required. In order to determine which algorithm is best, the designer should study the type of object-models that have to be transformed .

4. How to create specific transformations - In some cases the generic transformations are too general and the user of the transformer has to build specific transformations. The user should be able to build these transformations in the transform description language and be able to modify or reuse the generic transformations or to compose specific transformations out of the generic transformations.

In the general object-model transformer prototype implemented using SOFTWARE REFINERY, the generic transformations were implemented as functions. The object-model descriptions were designed to be written in ODL and parsed into the transformer where they are stored as abstract syntax trees. The transformer incorporates a variation of a top-down control algorithm and queries the user for any information which it can not infer for itself. Any model-specific transformations have to be implemented as functions and the function call has to be added to the controller. This limited prototype demonstrates the feasibility of constructing a general object-model transformer and provides a template which a designer can use to develop a transformer for a different environment.

The general object-model transformer works well for cases in which the source and target object-models are nearly equivalent and the differences between the models can be described by regular patterns. However in cases where the two object-models are significantly different, users have to write specific transformations that incorporate specific control algorithms. In these cases, the general transformations may not be used at all.

6.2 Recommendations for Further Research

- *Design and implement the control algorithm/controller developer.* Developing the controller which incorporates the control algorithm and specifies which transformations to apply is the most difficult part of the object-model transformer. A controller developer that built a controller for each type of object-model transformation would make a general object-model transformer more automatic and easier to use. Basically, it would decide which control algorithm to use given the source and target object-model descriptions and the transformation kernel. As stated in Chapter III, this developer would have to incorporate some type of automated inferencing technique based on a knowledge base of decision rules about object-model transformations.

- *Develop a standard for the transformation description language.* A transformation description language that provides generic commands for object manipulation and accessing the object-model descriptions would enhance the general object-model transformer. For this prototype REFINE was used as the transformation description language, but using REFINE forces the designer to handle the interfacing details between the transformer and the source and target environments. A better transformation description language abstracts the interface details away allowing the designer to specify the transformations and let the interface binding handle the details. A standard set of commands and the semantics of these commands should be developed. Then bindings from the transformation description language to any appropriate language can be developed.

- *Implement the ODMG ODL parser.* Since ODL is poised to become the standard language for describing ODBMS schemas, and since most object-model transformations involve some type of ODBMS, ODL seems like a logical choice for describing object-models to the transformer. In order for the transformer to use this description, some type of ODL parser needs to be developed. For the prototype object-model transformer, an ODL parser could be implemented using the DIALECT parser builder which is part of the SOFTWARE REFINERY environment. For other environments, common parser building tools could be used.

- *Design a windowing interface for the user queries.* The prototype user interface is extremely error-prone because a user has to manually enter the responses to any queries even though he is presented with a menu of options. This interface would be more effective and user friendly if the system presented the user with the menu in a pop-up window where he could use a mouse to click on the correct answer.

- *Research and develop different useful control algorithms.* Currently, the prototype has one control algorithm incorporated within its controller. But this control algorithm does not work for all instances of object-model transformations. There are other control algorithms, some that may be a superset of the one developed in this research, that are required to transform other models. These other algorithms need to be developed so that they can be added to the general object-model transformer. Then the transformer can select the best algorithm for each object-model transformation.

Appendix A. Script for the Prototype General Object-Model Transformer

A.1 Introduction

This section provides a script for a sample transformation. An instance of the School Model, described by Figure A.1, is transformed from the ITASCA ODBMS into the SOFTWARE REFINERY environment.

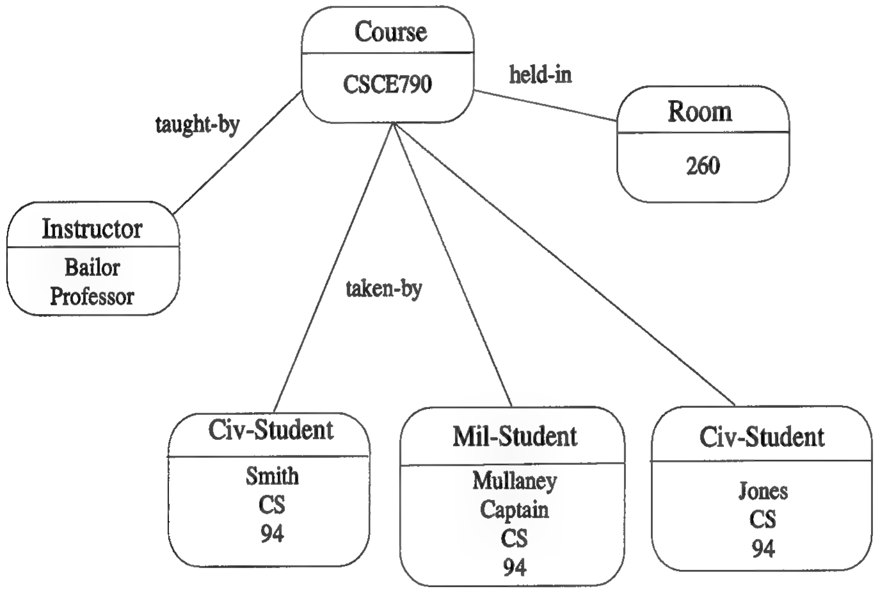

Figure A.1 Instance of a Course to be transformed

A.2 Script

At the REFINE prompt, the user should type:

```
.> (load "start-api-21")
```

This will connect the SOFTWARE REFINERY environment with the ITASCA ODBMS via the ITASCA Lisp API. At the next prompt, type:

```
.> (load "read-utilities")
```

This loads a set of input/output utilities. Next load the object-model for the target object-model which is a REFINE AST. Do this by typing:

```
.> (load "school-model")
```

Then load the source and target object-model descriptions by typing:

```
.> (load "domain-model")
```

Note that *domain-model* and *school-model* are files that would change if a different domain was being transformed. At the next prompt, load the object-model tranformations by typing:

```
.> (load "obj-trans")
```

Then load the controller by typing:

```
.> (load "controller")
```

Now that everything is loaded, it is time to execute the controller. Use the following command, where source-obj and target-obj are the descriptions for the source and target object-models.

```
.> (controller source-obj target-obj)
```

After this command the user will be presented with the following menu and query:

```
Entering Controller

 CIV-STUDENT

 MIL-STUDENT

 TEACHER

 ROOM
 COURSE

Which class should I transform first?:
```

For this example, type **course**. Then the following menu will appear:

```
 CS700

Choose an item:
```

For this example, there is only one application in the ODBMS, so type **cs700**. Then the transformer presents the next query.

```
Which transform should I use for COURSE  :  (NIL):
```

Since there is not a unique transformation for type COURSE, the user should hit return. After this, the following menus will appear. The user should pick the appropriate item from the menu and type it in at the prompt. In order to transform the sample model the responses given should be entered.

```
 C-STUD

 M-STUD

 TCHR

 RM

 CRSE

Which class should I transform COURSE to? : crse


 NAME

Which of these attributes map to NAME? : name
```

```
 TAKEN-BY-CIV

 TAKEN-BY-MIL

 HELD

 TAUGHT

Which relations correspond to TAUGHT-BY? :  (NIL): taught

 :  (NIL):


Which transform should I use for TEACHER  :  (NIL):


 C-STUD

 M-STUD

 TCHR

 RM

 CRSE

Which class should I transform TEACHER to? : tchr


 NAME

 T-RANK

Which of these attributes map to NAME? : name


 NAME

 T-RANK

Which of these attributes map to RANK? : t-rank
```

```
 TAKEN-BY-CIV

 TAKEN-BY-MIL

 HELD

 TAUGHT

Which relations correspond to HELD-IN? :  (NIL): held

 :  (NIL):


Which transform should I use for ROOM  :  (NIL):



 C-STUD

 M-STUD

 TCHR

 RM

 CRSE

Which class should I transform ROOM to? : rm


 RM-NUM

Which of these attributes map to NUMBER? : rm-num


 TAKEN-BY-CIV

 TAKEN-BY-MIL
 HELD

 TAUGHT

Which relations correspond to TAKEN-BY? :  (NIL): taken-by-civ

 :  (NIL): taken-by-mil

 :  (NIL):


Which transform should I use for MIL-STUDENT  :  (NIL):


 C-STUD

 M-STUD

 TCHR

 RM

 CRSE

Which class should I transform MIL-STUDENT to? : m-stud


 NAME

 M-STUD-RANK

 M-STUD-MAJOR

 M-STUD-YR

Which of these attributes map to NAME? : name


 NAME

 M-STUD-RANK
 M-STUD-MAJOR

 M-STUD-YR

Which of these attributes map to RANK? : m-stud-rank


 NAME

 M-STUD-RANK

 M-STUD-MAJOR

 M-STUD-YR

Which of these attributes map to MAJOR? : m-stud-major


 NAME

 M-STUD-RANK

 M-STUD-MAJOR

 M-STUD-YR

Which of these attributes map to YR? : m-stud-yr


Which transform should I use for CIV-STUDENT  :  (NIL):


 C-STUD

 M-STUD

 TCHR

 RM

 CRSE
```

```
Which class should I transform CIV-STUDENT to? : c-stud


 NAME

 C-STUD-MAJOR

 C-STUD-YR

Which of these attributes map to NAME? : name


 NAME

 C-STUD-MAJOR

 C-STUD-YR

Which of these attributes map to MAJOR? : c-stud-major


 NAME

 C-STUD-MAJOR

 C-STUD-YR

Which of these attributes map to YR? : c-stud-yr


Which transform should I use for CIV-STUDENT  :  (NIL):


 C-STUD

 M-STUD

 TCHR

 RM
 CRSE

Which class should I transform CIV-STUDENT to? : c-stud


 NAME

 C-STUD-MAJOR

 C-STUD-YR

Which of these attributes map to NAME? : name


 NAME

 C-STUD-MAJOR

 C-STUD-YR

Which of these attributes map to MAJOR? : c-stud-major


 NAME

 C-STUD-MAJOR

 C-STUD-YR

Which of these attributes map to YR? : c-stud-yr
```

As a result of the following sequence of choices the transformer has transformed the instance of a COURSE from the ODBMS into the SOFTWARE REFINERY environment. The following object description which corresponds to the object of COURSE CS700 is returned as a result of the transformation.

```
  #1<CS700 - a crse>
   re::class:  CRSE
```

```
name:  CS700
taught: {#2<BAILOR - a tchr>}
held: #3<a rm>
taken-by-civ: {#4<SMITH - a c-stud>, #5<JONES - a c-stud>}
taken-by-mil: {#6<MULLANEY - a m-stud>}
```

Bibliography

1. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures in Support of a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, Air Force Institute of Technology, December 1992.
2. Atwood, Tom and others. *The Object Database Standard: ODMG-93.* San Mateo, CA: Morgan Kaufmann Publishers, 1994.
3. Bailor, Major Paul D., "CSCE 793 - Formal-Based Methods in Software Engineering." Class Notes, 1993.
4. Bailor, Paul D. and others. "An Integrated Technology Approach to the Development of Software Composition Systems." *Computers in Engineering Conference.* (in press). 1995.
5. Beeri, Catriel. "Formal Models for Object Oriented Databases." *Deductive and Object-Oriented Databases* 405–430, Elsevier Science Publishers B.V. (North-Holland), 1990.
6. Cecil, Danny A. and Joseph A. Fullenkamp. *Using Database Technology to Support Domain-Oriented Application Composition Systems.* MS thesis, AFIT/GCS/ENG/93D-03, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.
7. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-04, Air Force Institute of Technology, December 1993.
8. Kim, Won. "Observations on the ODMG-93 Proposal for an Object-Oriented Database Language," *Sigmod Record*, *23*:4–9 (March 1994).
9. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.
10. Randour, Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base.* MS thesis, AFIT/GCS/ENG/92D-13, Air Force Institute of Technology, December 1992.
11. Reasoning Systems, Inc. *REFINE User's Guide.* Palo Alto, CA, May 1990.
12. Rumbaugh, James and others. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.
13. Silberschatz, Avi, et al. "Database Systems: Achievements and Opportunities," *Communications of the ACM*, *34*:110–120 (October 1991).
14. Soley, Richard Mark. "Object Model for Integration," *Standards and Interfaces*, *15*:149–166 (July 1993).
15. Weide, Timothy. *Development of a Visual System for a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/93M-05, Air Force Institute of Technology, March 1993.

Vita

Captain John P. Mullaney was born January 25, 1966 in Stuttgart-Bad Cannstatt, West Germany and graduated from Colonial Heights High School in Colonial Heights, Virginia in 1984. Upon graduation, he received an Air Force Reserve Officer Training Corps scholarship to attend the University of Notre Dame where he received a Bachelor of Science in Electrical Engineering and was commissioned in the United States Air Force in 1988. After attending the Communications-Computer Systems Engineer Officer School at Keesler AFB, Mississippi from February to May 1989, he served as a test engineer and test director with the Air Force Communications Command Operational Test and Evaluation Center at Wright-Patterson AFB, Ohio. In April 1991, he moved to Scott AFB, Illinois where he worked as a test engineer and software test manager for the Technology Integration Center and finally for Detachment 1 of the Air Force Operational Test and Evaluation Center. In April 1993, he entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio to pursue a Master of Science in Computer Science with an emphasis in Software Engineering.

Permanent address: 1215 Pleasant Dale Ave, Colonial Heights, VA 23834